"C++: 31 hours long video of freecodecamp"

Features:

supports data abstruction, means Data abstraction is a fundamental concept in computer science and tsoftware engineering that involves hiding the implementation details of data types and exposing only the essential features or behaviour to the outside world. Data abstraction is a powerful concept that facilitates the design, implementation, and maintenance of complex software systems by providing clear interfaces, encapsulating implementation details, promoting modularity, and supporting information hiding.

Statement: A statement is a basic unit of computation in a C++ program. Ends with a semicolon(;).

They are executed from top to bottom when program is run.

---

stream	Purpose
std::cout	Printing data to the console(terminal)
std::cin	Reading data from the terminal
std::cerr	Printing errors to the console
std::clog	Printing log messages to the console

---

input For Space separated word

```
int age;
std::string full_name;

std::cout << "Please type in your full name : " << std::endl;
std::getline(std::cin,full_name);
```

Variable: *A piece of memory that used to store specific type of data.*

Number System representation:

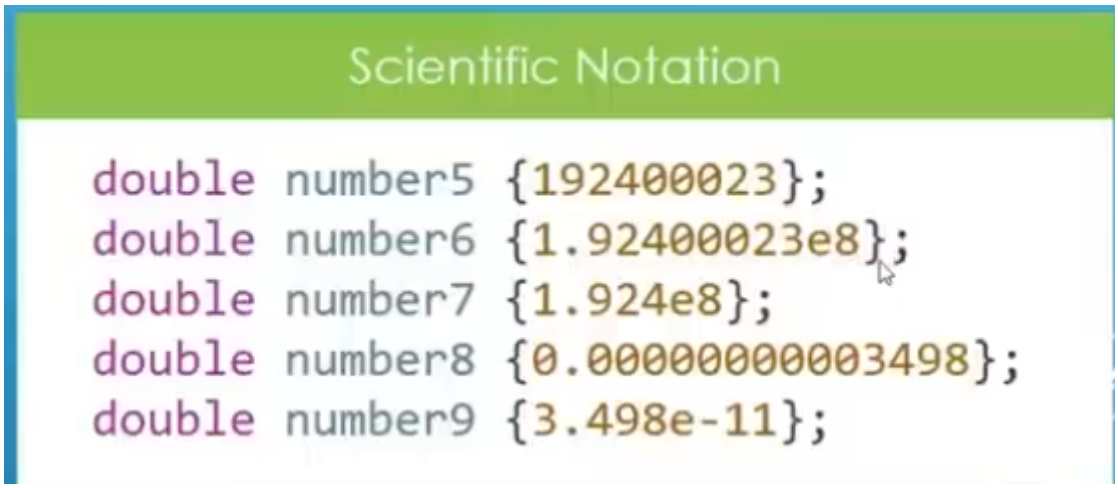
```
int number1 = 15; //Decimal
int number2 = 017;//Octal.
int number3 = 0x0f; //Hexadecimal
int number4 = 0b00001111; //Binary - C++14
```

**** Modifier like signed, unsigned used integral data type as decimal number/ whole number. Can't use for float, double(2.0, 2.5).*

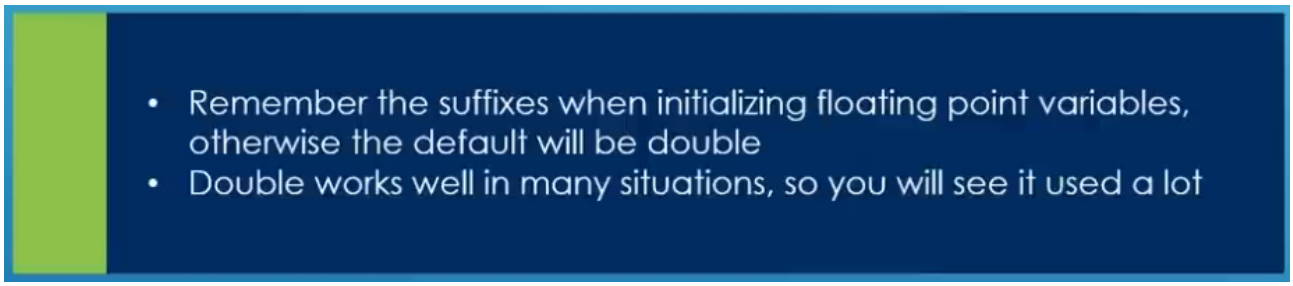
Float and Double:

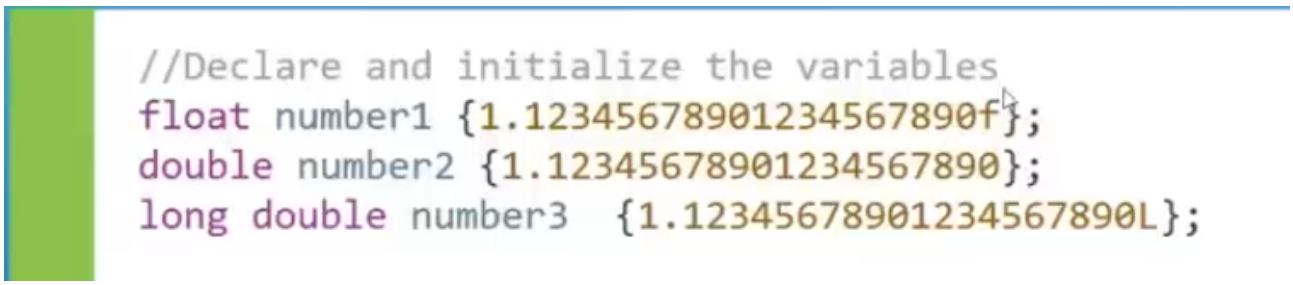
Type	Size	Precision	Comment
float	4	7	-
double	8	15	Recommended default
long double	12	> double	

***Precision includes digits before decimal point(.). 12547.325-> 12547 included in the precision.*
For Float: 12345.6789-> 89will be garbage value. Cause precision 7 for Float. May print like: 12345.6725
*** float a=123456789-> 89 will b garbage value. May print like: 12345.6745*

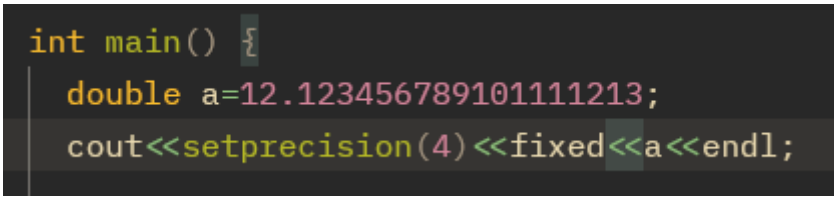


Floatiing point number memory representation is not similar to deciaml number. It explained in IEEE_754 -> IEEE_754-> Press ctrl & Click





cout<<setprecision(20);



If we not set precision, the default precision will be 6 digits.(Total digits. Not after decimal point)
*if we use **"fixed"** the with set precision(4), precision will be 4 after decimal point. Above a=12.1234;(ans)*
But not using "fixed" precision will be 4 for total digits. Above a=12.12;(ans)
used for set precision. Library> #include<iomanip>

***suffixes(like f, L):** u // unsigned*
ul // unsigned long
ll // long long

----------xxx----------

***Booleans** occupy **1byte/8bits** in memory.*
bool x=true; // or bool x=1;
bool y=false; // or bool y=0;
cout<<boolalpha; //used for printf true/false instead of 0/1
cout<<x<<" "<<y;
will print true, false.

Character:

```
char value = 65 ; // ASCII character code for 'A'
std::cout << "value : " << value << std::endl;
std::cout << "value(int) : " << static_cast<int>(value) << std::endl;
```

- ASCII was among the first encodings to represent text in a computer.
- It falls short when it comes to representing languages other than English and a few western languages. Think Arabic, East Asian Languages like Japanese, Chinese ,...
- There are better ways to represent text that is meant to be seen in different languages, one of the most common being Unicode
- The details of Unicode are out of scope for this course, just know that it's a robust way to represent text in different languages for a computer

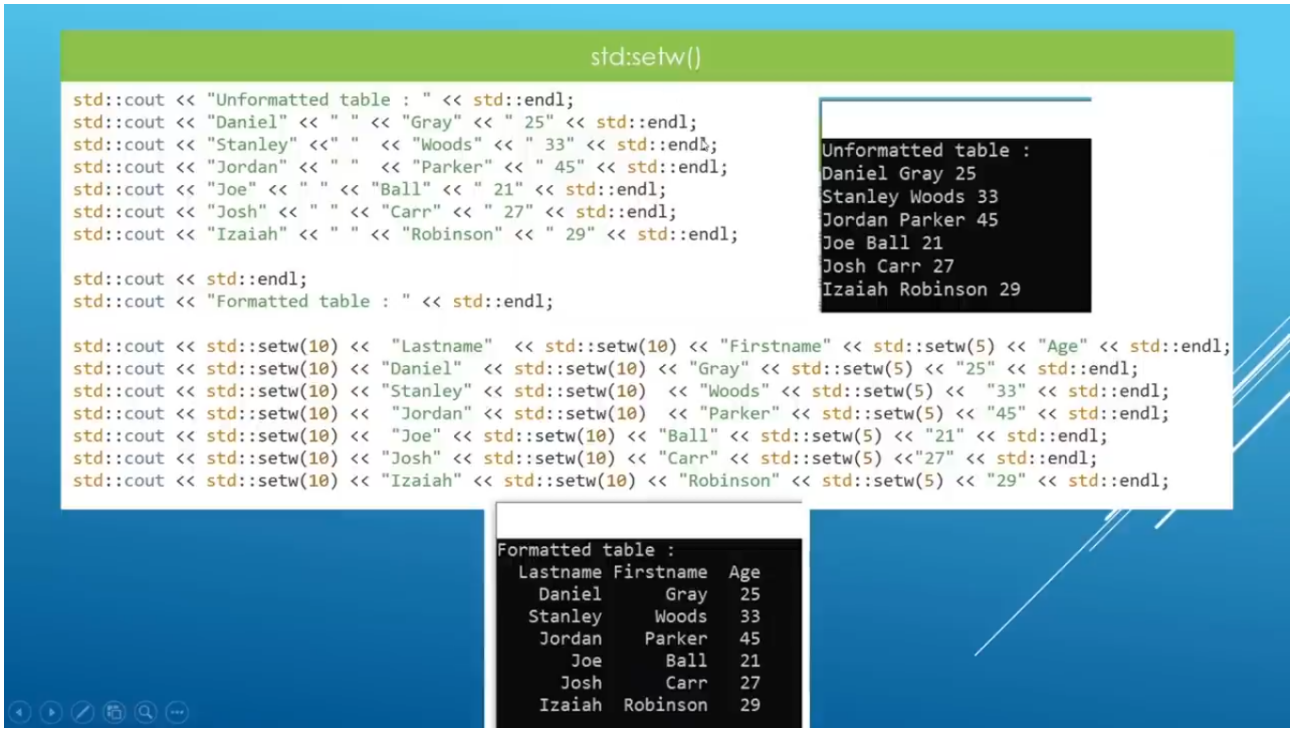
---

Precedence : which operation to do first
Associativity : which direction or which order

*** 31/10 means how many times 10 is gonna fit in 31. so ans is 3.*
** Relation Operator: > ,< , >=, <=*
** Logical Operator: &&, ||, !*

Manipulator: https://en.cppreference.com/w/cpp/io/manip *//for more*

** **std:flush:** when we print something, it does not go directly to the terminal. It store somewhere called "buffer". When buffer is full/ complete it goes to terminal. If use std:flush data directly goes to console/terminal instead of goes to buffer.*



*****setw() :*** *set width*

cout<<right; // printf from right. *Left for left alinement*

Head	Head
1	2

cout<<setfill('-') // blank space fill with '-'

```
Lastname----------Firstname----------Age-------
Daniel------------Gray----------------25--------
Stanley-----------Woods---------------33--------
Jordan------------Parker--------------45--------
Joe---------------Ball----------------21--------
Josh--------------Carr----------------27--------
Izaiah------------Robinson------------29--------
```

*for finding max min numeric number limits data type can hold.
Need this> #include<limits>
https://en.cppreference.com/w/cpp/types/numeric_limits

```cpp
main.cpp > main()
#include <iostream>
#include<iomanip>
#include<limits>
using namespace std;
#define NL printf("\n")
#define nl endl

int main(){
    int a=300, b=100;
    cout<<numeric_limits<int>::min()<<nl;
    cout<<numeric_limits<int>::max()<<nl;
    NL;
    cout<<numeric_limits<unsigned int>::min()<<nl;
    cout<<numeric_limits<unsigned int>::max()<<nl;
}
```

----------x----------

#include<cmath> : abs(), pow(), ceil(), log(), sqrt(), sin(), tan() etc
https://en.cppreference.com/w/cpp/header/cmath
for log(): log(10) means $\log_e(10)$. So have to fix the base as $\log_{10}(10)$. E=2.71..
* round(): 3.5 will make 4, and 3.49 will make 3.

Integral types less than 4 bytes in size don't support arithmetic operations

if we take data type less than 4 byte and perform arithmatic operation compiler automatically convert it to 4 byte. This behavior also present on other operator like bitwise operator.(>>, <<)

---

flow control: if else, switch, ternary operator.
***Switch:** if we not use "Break", the case which match, after that all case will execute and print every case value.
* we can use int, char , double, **enum** etc but not string as case.

---

* If we use "const" before array. We cant modify array elements.
* a[ ]={2,3,7,5,2,3}; size(a) return the size of array. /Or sizeof(a)/sizeof(a[0]);

Enum: An **enum** is a special type that represents a group of constants (unchangeable values).
To create an enum, use the enum keyword, followed by the name of the enum, and separate the enum items with a comma.

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
};
```

Enum is short for "enumerations", which means "specifically listed".
To access the enum, you must create a variable of it.
Inside the main() method, specify the enum keyword, followed by the name of the enum (Level) and then the name of the enum variable (myVar in this example):

```
enum Level myVar;
```

By default, the first item (LOW) has the value 0, the second (MEDIUM) has the value 1, etc.
If you now try to print myVar, it will output 1, which represents MEDIUM:

```
int main() {
  // Create an enum variable and assign a value to it
  enum Level myVar = MEDIUM;

  // Print the enum variable
  cout << myVar;

  return 0;
}
```

As you know, the first item of an enum has the value 0. The second has the value 1, and so on. To make more sense of the values, you can easily change them:

```
enum Level {
  LOW = 25,
  MEDIUM = 50,
  HIGH = 75
};
```

```
int main() {
  enum Level myVar = MEDIUM;
  cout << myVar;  // Now outputs 50
  return 0;
}
```

Note that if you assign a value to one specific item, the next items will update their numbers accordingly:

```
enum Level {
  LOW = 5,
  MEDIUM, // Now 6
  HIGH // Now 7
};
```

Example:

```
enum Level {
  LOW = 1,
  MEDIUM,
  HIGH
};

int main() {
  enum Level myVar = MEDIUM;

  switch (myVar) {
    case 1:
      cout << "Low Level";
      break;
    case 2:
      cout << "Medium level";
      break;
    case 3:
      cout << "High level";
      break;
  }
  return 0;
}
```

Why And When To Use Enums?

Enums are used to give names to constants, which makes the code easier to read and maintain.Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

Pointer

Pointer is special kind of variable.

int int_num{}; or int* int_num; // will point to a integer type variable //initialise with null*
// pointer(nulptr)

double frac_num{}; // will point to a double type variable*

*int * int_num{}; // this initialisation with {} is going to initialise with special address means it is not*
//pointing any variable. Means initialize with ***nullptr***

*int * int_num{nullptr}; // this pointer not pointer anywhere*

*** pointer of int, double, char etc all are same size. Cause they only store address.*

Char *ptr{"Hello world"}; *// Pointer will point to 1st character. Some compiler will not compile(MSVC). GCC will give warning and compile. A whole string is assigning to char type pointer.* ***See:10:17:00***

Cout<<ptr; // will print whole string.

*Cout<<*ptr;// print 1st character 'H';*

**ptr='B';// this may give error. Cause compiler will think it as const char array.*

If want to modify. Don't use character pointer, use array like: ***char msg[10]=="Hello world";***

or use ***const char *ptr{"Hello world"};*** *// no warning. Can't modify also.*

Dereferencing pointer: *reading something(value) on the address of the pointer. Cout<<*ptr;*

string with pointer: *char* p_msg= "Hello World!"; // the pointer will point to the 1st character of string*

this will compile with warning. Use **const char* p_msg= "Hello World!";*

*printf **p_msg** will print whole string. But using dereferencing will print 1st character(*p_msg).*

**** without **const** it gives warning/refuse to compile cause compiler is going to convert string into char array of constant char. What we are using is points to that is not a const char pointer. So pointer here might be used to try or modify data. That's why it refuse unless using **const.***

Check at 10:17min

*int *ptr;// contain junk address*

int a=12;

**ptr=&a;*

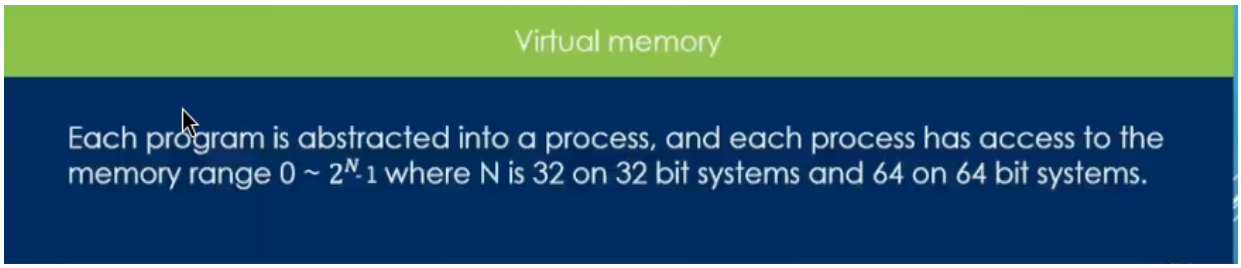
*uninitialized pointer contain junk address. Assigning a value to it(*ptr=12)May cause error. Could be point to a memory which is used by OS. May cause disaster.*

*Use this: int a; int *ptr=&a;*

```
//Initializing pointer to null
//int *p_number3{nullptr}; // Also works
int * p_number3 {}; // Initialized with pointer equivalent of zero : nullptr
                    // A pointer pointing nowhere
//*p_number3 = 33; // Writting into a pointer pointing nowhere : BAD, CRASH
```

Memory Map

When we run a program it runs on RAM. Various program of OS or other is running on memory.



This process thinks it own 0~2^N amount of memory which is virtual memory.

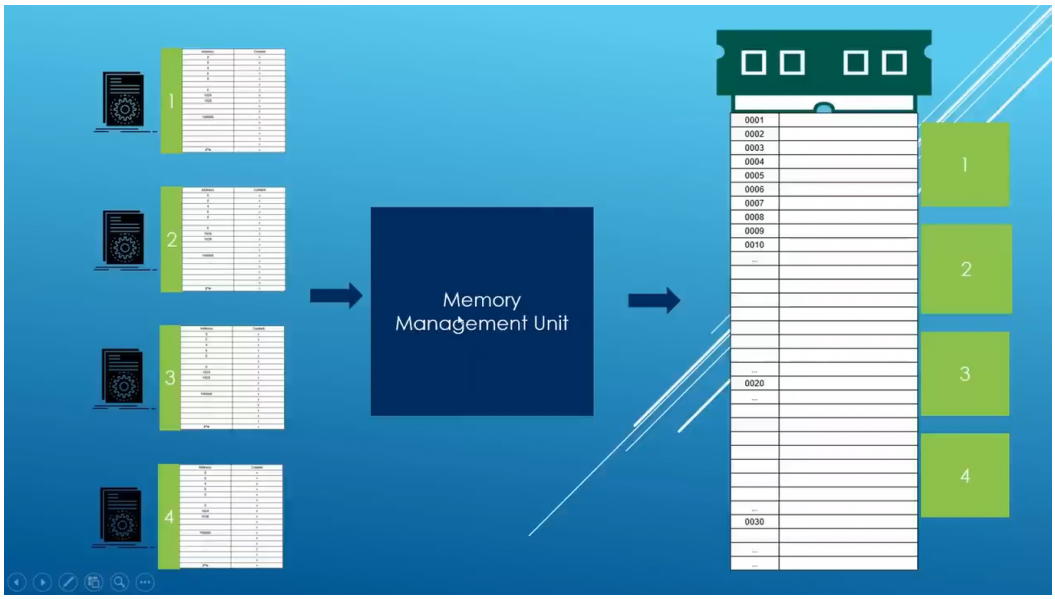
When we run a program it is going to go through a CPU section called memory management unit(MMU).

The entire program is not loaded in real memory by the CPU and MMU. Only parts that are about to be executed are loaded. Making effective use of real memory, a valuable and lacking resource.

Part that are likely not to be used are discarded from the RAM.

MMU really does is, helps us mapping between the memory map in ur program and the real thing we have in RAM.

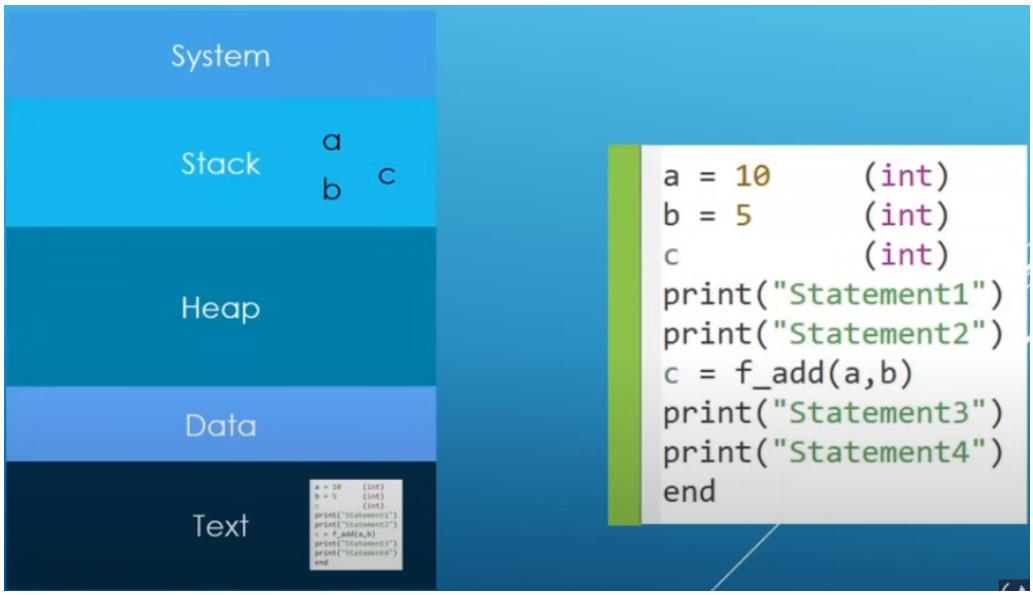
If we run few program, they are going to go through MMU and MMU is going assign them real section on RAM



** Since program thinks it 2^n -1 memory, The MMU is going to transform between the idea the program has and the RAM we have(assigned memory by MMU).*

** The memory map/Structure of program is standard format defined by OS. That's why we can't run directly window program on Linux.*

**Memory map is divided into a lot parts*



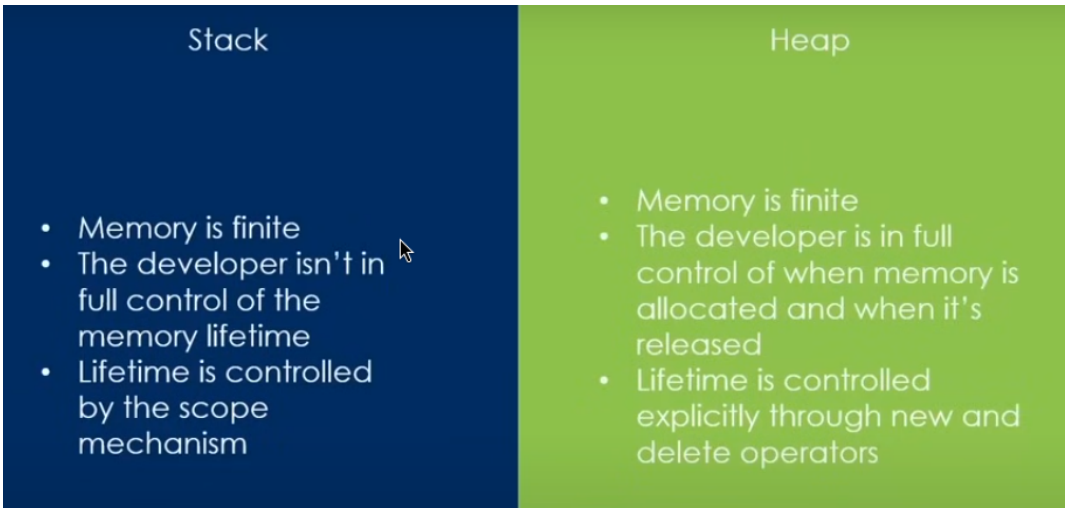
* ***STACK:*** *Local variable stores on stack section.*

    * *print, statement, function calls others store on stack section.*

* ***TEXT:****Actual binary load on Text so that CPU can execute it.*

* ***HEAP:****Additional memory when we run out of stack memory also to make things better for program, Used for run time.*

Dynamic Memory Allocation



2nd point- full Control: *when declare a variable a=23; in stack it remove when scope is over. Developer doesn't have full control. But in heap developer have full control when a variable comes to work and dies.*

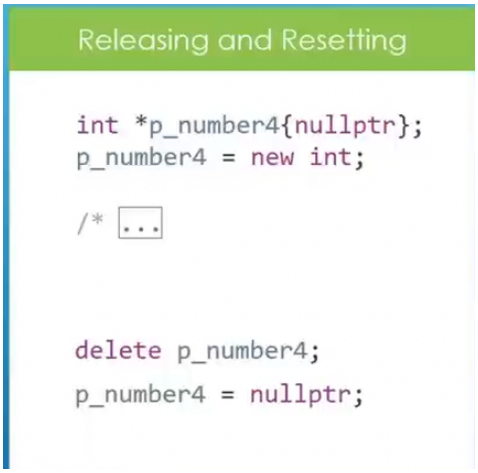
>> 10:41 min

Allocate dynamic memory through 'new'

```
//Initialize the pointer with dynamic memory. Dynamically allocate
//memory at run time and make a pointer point to it

int *p_number4{nullptr};
p_number4 = new int;     // Dynamically allocate space for a single int on the heap
                         // This memory belongs to our program from now on. The system
                         // can't use it for anything else, untill we return it.
                         //After this line executes, we will have a valid memory location
                         // allocated. The size of the allocated memory will be such that it
                         // can store the type pointed to by the pointer

*p_number4 = 77; // Writting into dynamically allocated memory
std::cout << std::endl;
std::cout << "Dynamically allocating memory : " << std::endl;
std::cout <<"*p_number4 : " << *p_number4 << std::endl;
```

**set up a pointer which point to heap memory.*

After initializing a pointer with nullptr. When p_number4=new int execute the OS is allocate a memory on heap. Variables are usually stores on stack section. It removes when variable containing scope will over. But if we allocate a memory on heap by "p_number4=new int;" . This memory will live though its scope is over. It will stay until return it to operating system. To return:



Use 'delete' to return memory to the operating system. After return reset pointer to 'nullptr' is good to make it clear that no valid data pointer is pointing.

** Using 'delete' remove the allocated heap memory which is pointed. Not the pointer. If pointing to 'nullptr', delete will do nothing.*

**** Always remember to release memory.*

*****Dangling Pointer:*** *A pointer that doesn't point to a valid memory address. Trying to dereferencing and using a dangling pointer will results in undefined behaviour.*

How dangling pointer create:

1. Uninitialized pointer.

2. Delete pointer

3. Multiple pointer points to same memory.

Solution:

1. Initialize pointer. Either with valid address or nullptr.

2. Reset pointer after delete.(Either with valid address or nullptr.)

3. For multiple pointer point to same address, make sure master pointer is clear/ reset.

**** Always check if pointer is nullptr or not by if-else.*

'New' fails:

When allocating an array with pointer with huge size(1000000000000000000). It may fails and program can crash. We can use ***exception mechanism or 'nothrow'*** *to prevent crashing.*

```
int *ptr=new int[100000000000];
for(size_t i=0; i<10000000; i++){
    int *ptr= new int[10000];
}
```

Both allocation (with for loop or without) fail.

** **Solve with 'exception mechanism':***

```
for(size_t i=0; i<100; i++){
  try {
    int *ptr= new int[1000000000000];
  } catch (exception& ex) {
      cout<<"Problem: "<<ex.what();NL;
  }
}
```

With 'exception' we can catch the problem. What is called 'handle' in the problem. Suppose we are going to set up color and color fails. UI(interface) may show black and white. But program will keep running ***"what()"*** *function will show the error.*

***with 'nothrow':** *If "new" fails, we are going to get "nullptr" stored.*

```
for(size_t i=0; i<100; i++){
  int *ptr= new(nothrow) int[1000000000000];
  if(ptr!=nullptr) cout<<"Memory allocated";
  else cout<<"Memory not allocated";
  NL;
}
```

For clear understanding see

https://www.youtube.com/watch?v=uoCuMTzD9AE&list=PLgH5QX0i9K3q0ZKeXtF--CZ0PdH1sSbYL&index=90 // anisul islam lecture 92. Exception handling

NULL pointer safety:

```
int *p_number{};//Initialized to nullptr
```

check if null or not.

```
if(!(p_number==nullptr)){
    std::cout << "p_number points to a VALID address : "<< p_number << std::endl;
}else{
    std::cout << "p_number points to an INVALID address." << std::endl;
}
```

Or,

```
if(p_number){
    std::cout << "p_number points to a VALID address : "<< p_number << std::endl;
}else{
    std::cout << "p_number points to an INVALID address." << std::endl;
}
```

Pointer address implicitly converted into boolean. (null==0).

** After use delete set pointer to nullptr for safety.*

----------x----------

Memory Leaks:

Reassignment of stack address to active dynamic address pointer

```
int *p_number {new int{67}}; // Points to some address, let's call that address1

//Should delete and reset here

int number{55}; // lives at address2

p_number = &number; // Now p_number points to address2 , but address1 is still in use by
                    // our program. But our program has lost access to that memory location.
                    //Memory has been leaked.
```

```
//Double allocation
int *p_number1 {new int{55}};

//Use the pointer

//Should delete and reset here.

p_number1 = new int{44}; // memory with int{55} leaked.
```

while 2nd allocation, pointer changes its pointing location to 2nd memory. Don't have access of 1st memory(55). Should delete 1st then allocate 2nd.

Pointer in local scope

```
#include <iostream>

int main(int argc, char **argv)
{

    ...

    {
        int *p_number2 {new int{57}};
    }
    //Memory with int{57} leaked.

    return 0;
}
```

After local scope is over, pointer is gonna die, but allocated memory will remain and loose access.

These memory leaks may causes program crash.

Dynamic array: *Array stores on the heap.*

Array dynamic allocation

```
size_t size{10};

//Different ways you can declare an array
//dynamically and how they are initialized

double *p_salaries { new double[size]}; // salaries array will
                                        //contain garbage  values
int *p_students { new(std::nothrow) int[size]{} }; // All values initialized to 0

double *p_scores { new(std::nothrow) double[size]{1,2,3,4,5}}; // Allocating memory space
                                                               // for an array  of size double
                                                               //vars. First 5 will be initialized
                                                               //with 1,2,3,4,5, and the
                                                               //rest will be 0's.
```

```
//nullptr check and use the allocated array
if(p_scores){
    //Print out elements. Can use regular array access notation, or pointer arithmetic
    for( size_t i{}; i < size ; ++i){
       std::cout << "value : " << p_scores[i] << " : " << *(p_scores + i) << std::endl;
    }
}
```

Release Memory:

```
delete[] p_scores;
p_scores = nullptr;

delete[] p_students;
p_students = nullptr;

delete[] p_salaries;
p_salaries = nullptr;
```

Dynamic arrays are created at run time not compile time.

Pointers and arrays are different

```
//Pointers initialized with dynamic arrays are different from arrays :
//std::size doesn't work on them, and they don't support range based for loops

double *temperatures = new double[size] {10.0,20.0,30.0,40.0,50.0,60.0,70.0,80.0,90.0,100.0};

//std::cout << "std::size(temperatures) : " << std::size(temperatures) << std::endl;//Error

//Error : temperatures doesn't have array properties that are needed for
// the range based for loop to work.
for (double temp : temperatures){
    std::cout << "temperature : " << temp << std::endl;
}

//We say that the dynamically allocated array has decayed into a pointer
```

Explanation:

You're creating a **pointer**, not a real array type. It points to a memory block of 5 integers on the heap — but the compiler **doesn't know the size** of that memory.

🚫 No Range-Based Loops with `new[]`

cpp Copy Edit

```cpp
int* arr = new int[5];
for (int x : arr)  // ❌ Error: pointer has no size info!
```

You must use:

cpp Copy Edit

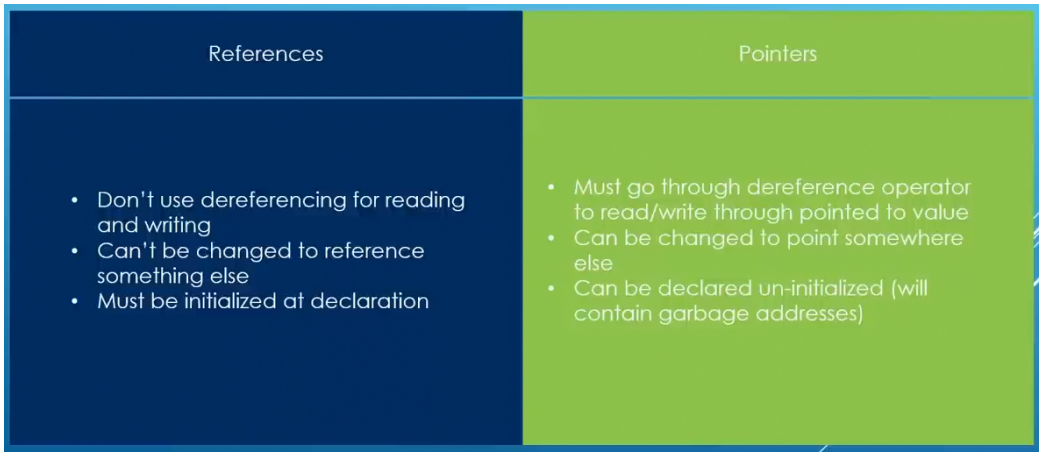
```cpp
for (int i = 0; i < 5; ++i)
    cout << arr[i];
```

Reference: *A reference is an alias (another name) for an existing variable. It does not create a new variable or occupy memory, it just gives another way to access the same memory.*

```
int s=10;
int &ref=s;
printf("%d %p %p %d\n", s, &s, &ref, ref);
ref=100;
printf("%d %d\n", s, ref);
s=1000;
printf("%d %d\n", s, ref);
```

If we change reference value or variable value, both contribute same changes.
*** "&ref" and "&s" both has same memory address.*

```cpp
//Declare pointer and reference

double double_value {12.34};

double& ref_double_value {double_value}; // Reference to double_value

double* p_double_value {&double_value}; //Pointer to double_value


//Reading
std::cout << "double_value : " << double_value << std::endl;
std::cout << "ref_double_value : " << ref_double_value << std::endl;
std::cout << "p_double_value : " << p_double_value << std::endl;
std::cout << "*p_double_value : " << *p_double_value << std::endl;
```



Reassign a Pointer to others variable, but reference can't.

```cpp
int s=10;
int &ref=s;
int ss=12;
ref=ss;
printf("%d %d\n", s, ref);
```

Here 'S and ref' value will changes to 12.
***Use case:** *if want to modify original variable inside a function. This save memories.*

```cpp
void square(int &x) {
    x = x * x;
}

int main() {
    int a = 5;
    square(a);  // modifies 'a'
    cout << a;  // Output: 25
}
```

** Return reference by function to a local or global variable.*

```
vector<int> v = {1, 2, 3};

for (int &x : v) {
    x += 10;  // modifies original vector
}
```

- Without `&`, you'd just be modifying a copy.

References are somewhat like const pointers

```
//References behave like constant pointers, but they have
//a much friendlier syntax as they don't require dereferencing
//to read and write through referenced data.

double *const const_p_double_value {&double_value};

const_p_double_value = &other_double_value;// Error
```

** Using "const" before reference makes the variable unchangeable. This constant only applies to reference. Not variable. Can't changes variable value with "const reference". But variable itself can changes its value.*

```
int s=10;
const int &ref=s;
int ss=12;
// ref=ss; // show error, cause "const"
s=12;
printf("%d %d\n", s, ref);
```

Value will changes to 12.

----------x----------

**** size(), sizeof() function return the size. But for character array it includes 'null'. But for string size() don't count null.*

**** strlen() is used for character array. Not for string.*

*char *a="asdf"; // this is string literal. Not modifiable.*

Char a[]="asdf"; // this is character array. Modifiable. Use stack memory.

Using "const" is safe in string literal.

****String literal is actually a character array. When we assign it to "const char * ", it automatically converts into a pointer to 1st element.*

*Const char *msg="Hello I am here.";*

Declaration	Modifiable?	Memory Type	Notes
`char* s = "text"`	❌ No	Read-only segment	**Undefined behavior** to change
`const char* s = "text"`	❌ No	Read-only segment	✅ Safe and correct
`char s[] = "text"`	✅ Yes	Stack	Literal copied into array
`std::string s = "text"`	✅ Yes	Heap	Best practice in C++

Character array lives on read only memory. Can't modify.

----------x--------

swap 2 number.

1. a=a+b; b=a-b; a=a-b;

2. a=a^b; b=a^b; a=a^b;

----------x----------

Returning from function as Reference: *Usually function returns values(int, char etc). But sometime compilers are smart enough that that return reference instead of values. Avoid copies. See below*

In modern compilers, return by value is commonly optimized out by the compiler when possible and the function is modified behind your back to return by reference, avoiding unnecessary copies!

Example:

```
string add_str(string s1,string s2){
  string ress=s1+s2;
  cout<<ress<<" "<<&ress;NL;
  return ress;
}

int main(){
  string s1="Hello ", s2="World!";
  string res=add_str(s1,s2);
  cout<<res<<" "<<&res;NL;
```

Output:

```
› g++ proj.cpp && ./a.out
Hello World! 0x7ffe27076d50
Hello World! 0x7ffe27076d50
```

Here both addresses are same. Its returns the reference. Using reference mechanism.

----------x----------

Function Overloaded: *Means we can declare multiple function with same name in the same scope, but with different parameter lists. Like parameter type(int, double, float).*

Below All allowed.

```
int max(int a, int b){
  int sum=a+b;
  return sum;
}
double max(double a, int b){
  double sum=a+b;
  return sum;
}
```

```
int max(int a, int b){
  int sum=a+b;
  return sum;
}
int max(double a, int b){
  double sum=a+b;
  return sum;
}
```

```
int max(int a, int b, int c){
  int sum=a+b+c;
  return sum;
}
int max(int aa, int bb){
  double sum=aa+bb;
  return sum;
}
```

Not allowed.(below)

```
int max(int a, int b){
  int sum=a+b;
  return sum;
}
int max(int a, int b){
  double sum=a+b;
  return sum;
}
```

```
int max(int a, int b){
  int sum=a+b;
  return sum;
}
int max(int aa, int bb){
  double sum=aa+bb;
  return sum;
}
```

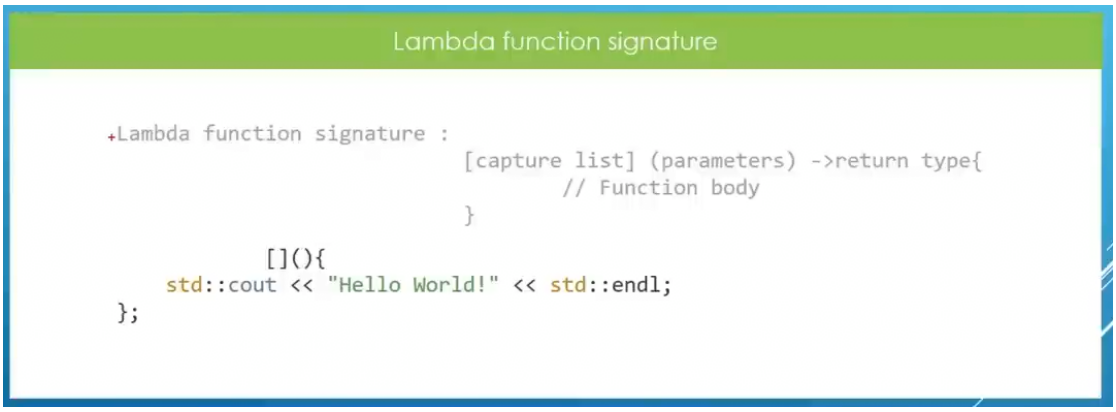
```cpp
int max(int a, int b){
    std::cout << "int overload called" << std::endl;
    return (a>b)? a : b;
}

double max(double a, double b){
    std::cout << "double overload called" << std::endl;
    return (a>b)? a : b;
}
```

*When **'int'** type variable is passed as argument "int overload will called". Same as double and others.*

----------x----------

lambda Function:

A mechanism to set up anonymous functions (without names). Once we have them set up, we can either give them names and call them , or we can even get them to do things directly.



** Return type is not important. If keep blank, compiler is gonna deduce its type by itself.*

** Use ';' after function body. Because lambda function is a statement.*

```cpp
int main(){
    auto fun=[](int a, int b)->int{
        printf("jhbgvf\n");
        return a+b;
    };
    printf("%d\n", fun(2,1));
    //Or call without name below
    [](){
        printf("sdfg");
    }();

}
```

```
[](double a, double b){
    std::cout << "a + b : " << (a + b) <<  std::endl;
}(12.1,5.7);
```

```
int main(){
  auto fun=[](int a, int b)->int{
    printf("jhbgvf\n");
    return a+b;
  }(2,1);
  printf("%d\n", fun);
```

Here if lambda function return something, it going to assign to variable 'fun'.

Capture lists on lambda function: *Capture list is a part of lambda function inside the square brackets which tells the lambda function, which variable from the surrounding scope it can use and how(like using a copy of a variable or references).*

** When a lambda function capture values, it made a copy of that variable on the memory. So if that variable is changed later it will not effect on that lambda function. Lambda function retain the old value unless we use variable as reference. See below...*

```
int a=12;
auto fun=[a](){
  printf("Inner value: %d\n", a);
};
for(int i=0; i<5; i++){
  printf("Outer value: %d\n", a);
  fun();
  a++;
}
```

```
› g++ proj.cpp && ./a.out
Outer value: 12
Inner value: 12
Outer value: 13
Inner value: 12
Outer value: 14
Inner value: 12
```

See when we use variable as reference:

```
int a=12;
auto fun=[&a](){
  printf("Inner value: %d\n", a);
};
for(int i=0; i<3; i++){
  printf("Outer value: %d\n", a);
  fun();
  a++;
}
```

```
› g++ proj.cpp && ./a.out
Outer value: 12
Inner value: 12
Outer value: 13
Inner value: 13
Outer value: 14
Inner value: 14
```

If we print the addresses, we can clearly see that both(variable a, and lambda function variable a) variable have different addresses.

**** "[=]" using this as capture list it will grab all variable from the surrounding scope. (To capture value)*

**** "[#]" using this as capture list it will grab all variable from the surrounding scope. (To capture as references)*

**** using reference, all have same addresses.*

----------x----------

Function Template:

Function Template by value: *Function Template is a mechanism in c++ to set up a blueprint for functions, But compiler going generate the actual code when it sees the function called. Means to avoid code repetition.*

```
int max(int a, int b){
    return (a>b)? a : b;
}

double max(double a, double b){
    return (a>b)? a : b;
}
```

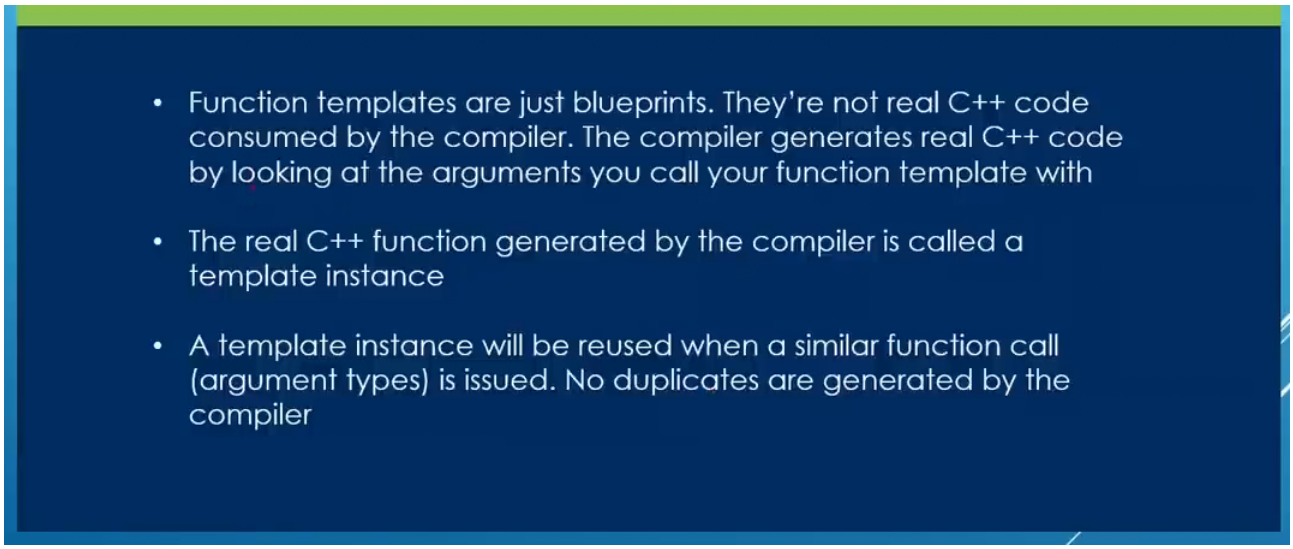
Here there are multiple function overload. They are doing the work. To minimise this type of multiple overload of same work. We can use Template.

```
template <typename T> T maximum(T a,T b);

int main(int argc, char **argv)
{
    int a{10};
    int b{23};

    double c{34.7};
    double d{23.4};

    std::string e{"hello"};
    std::string f{"world"};

    std::cout << "max(int) : " << maximum(a,b) << std::endl; // int version created
    std::cout << "max(double) : " << maximum(c,d) << std::endl;// double version created
    std::cout << "max(string) : " << maximum(e,f) << std::endl;// string version created

    /* ...

    return 0;
}

template <typename T> T maximum(T a,T b){
    return (a > b) ? a : b ; // a and b must support the > operator. Otherwise, hard ERROR
}
```

" T " is a place holder for the data type function is going to receive as argument. But all argument data type ***have to be same(there is a room for not same data type)****. When compiler see, template function called. It going to look at the data type of arguments which are passing to function. Then compiler generate same function with that data type only when the function is called .*

*** can also pass string by argument.*

Function templates are not c++ code. It just a function blueprint.



If data types are not same passing as argument, we can explicitly set the type with<double>. This basically tells the compiler to generate double/int etc. template instance function for this calling. And implicitly convert other type to determined type. In below example "a" variable int type and will convert to double.

*** We can see this internal conversion of function template on* ***cppinsights.io.***

```
template<typename T> T maxii(T a, T b){
  return a+b;
}

int main(){
  int a=12;
  int b=34;
  double c=12.22;
  auto re=maxii<double>(a,c);
  cout<<re;
```

This will give no error. Cause we explicitly tell to generate a double type template instance function. Otherwise, different type will not accept. Will throw an error.

If we use sizeof() to see size of the "re" variable. We can see is it double, int etc.

Template type parameter by references: *Recall references procedure, template procedure. All same concept.*

```
template<typename T> T maxii(const T& a,const T& b){
  return a>b?a:b;
}

int main(){
  int a=12;
  int b=34;
  int re=maxii(a,b);
  cout<<re<<endl;

}
```

```
template <typename T> const T& maximum(const T& a, const T& b); // Declaration
template <typename T> T maximum(T a, T b); // Declaration
```

function overloaded. Cause calling

maximum(a,b); // this is used for both template with value and with reference. That's why get confused.

----------x----------

Template Specialization: *This is specially for "const char pointer" like:*

const char* x="asdf";

for const char pointer regular method will not work. There is a special template mechanism for it.

```
//Template specialization
template <>
const char* maximum<const char*> (const char* a , const char* b);
```

This does not compare like others in template. See 18:50 Hr

Instead it use c++ template library function "strcmp" to compare. See on www.cppreference.com about strcmp.

In order to use template specialization we have to declare primary template like below…

Specializing maximum for const char*

```
template <>
const char* maximum<const char*> (const char* a , const char* b){
    //std::strcmp doc : https://en.cppreference.com/w/c/string/byte/strcmp
    return (std::strcmp(a,b) > 0) ? a : b ;
}
```

Return value

Negative value if lhs appears before rhs in lexicographical order.

Zero if lhs and rhs compare equal.

Positive value if lhs appears after rhs in lexicographical order.

In order to use template specialization we have to declare primary template like below…

```
template<typename T> T maxii(const T a,const T b){
  return a>b?a:b;
}
```

Whole code:

```cpp
template<typename T> T maxii(const T a,const T b){
  return a>b?a:b;
}
template<>
const char* maxii<const char*>(const char* a, const char* b){
  return (strcmp(a,b)>0)?a:b;
}
int main(){
  int a=12;
  int b=34;
  const char* x="dasdd";
  const char* y="ebaaa";
  const char* re=maxii(x,y);
  cout<<re<<endl;
}
```

C++ 20:

Concept: *Concept is a mechanism to set up constrain or restriction on template parameter of our function template. For example we can set that function to be called only integer and we call it something which isn't ans integer, it will give a compiler error.*

```cpp
template<typename T>
requires integral<T>
T add(const T a,const T b){
  return a+b;
}

int main(){
  int a=98, b=100;
  auto rs=add(a,b);
  cout<<rs;
}
```

If we set double instead of int, this will compiler error. Cause we set "integral" type to accept as argument in function template.

These concepts are introduced in c++20. So use "-std=c++20" during compiling.

```
~/code via C v15.2.1-gcc
› g++ -std=c++20 proj.cpp  && ./a.out
```

Concepts are introduced in c++20. Before that we can use this(below) inside function template. And set a custom message if condition does not meet.

```cpp
static_assert(std::is_integral<T>::value , "Must pass in an integral argument");
```

Type Traits: *Intro on C++11. A type trait is a small template "tool" that tells you something about a type, or transforms a type, at compile time. On above we are checking "T" is an integral or not by "is_integral<>". Boolean value. Also Use this as requires. See below*

Syntax1 : using type traits

```
template <typename T>
requires std::is_integral_v<T>// Using a type trait
T add (T a, T b){
    return a + b;
}
```

*We can use type traits on **requires**.*

```
template<typename T>
requires is_floating_point_v<T>
T add(T a,T b){
  return a+b;
}

int main(){
  double a=12.5;
  double b=12;
  auto rs=add(a,b);
  cout<<rs;
}
```

Some Type Traits:

is_void (C++11)	checks if a type is `void` (class template)
is_null_pointer (C++11)(DR*)	checks if a type is `std::nullptr_t` (class template)
is_integral (C++11)	checks if a type is an integral type (class template)
is_floating_point (C++11)	checks if a type is a floating-point type (class template)
is_array (C++11)	checks if a type is an array type (class template)
is_enum (C++11)	checks if a type is an enumeration type (class template)
is_union (C++11)	checks if a type is a union type (class template)
is_class (C++11)	checks if a type is a non-union class type (class template)
is_function (C++11)	checks if a type is a function type (class template)
is_pointer (C++11)	checks if a type is a pointer type (class template)

Some ways to declare concepts:

Syntax1

```
template <typename T>
requires std::integral<T>
T add (T a, T b){
    return a + b;
}
```

Syntax4

```
template <typename T>
T add (T a, T b) requires std::integral<T>{
    return a + b;
}
```

Syntax 3 is only allowed for, when we use "auto".(Below)

Syntax3

```
auto add (std::integral auto a,std::integral auto b){
    return a + b;
}
```

Syntax2

```
template <std::integral T>
T add (T a, T b){
    return a + b;
}
```

Example for Type Trait:

A type trait is a template utility in the C++ standard library (in <type_traits>) that allows you to query information about a type or transform a type at compile time.

Think of type traits as little "questions" or "tools" about types:

1. Is this type an integer?

2. Is this type const-qualified?

3. What happens if I remove a pointer from this type?

4. Are these two types the same?

```
template <typename T>
void print(const T& value) {
    if constexpr (std::is_integral<T>::value) {
        cout << value << " is an integral type\n";
    } else {
        cout << value << " is NOT an integral type\n";
    }
}

int main() {
    print(42);     // integral
    print(3.14);   // not integral
}
```

Here the compiler removes the unused branch at compile time → no runtime cost.

Here without "constrxpr" if is checked at run time. So both branch(if, else) must compile because compiler doesn't know which condition will be true.

But with "constexpr"

** if constexpr means the compiler chooses the branch at compile time.*

** The unused branch is discarded before code generation — it's as if it never existed. Means for ("print(42)") compiler will keep only if part. And for ("print(3.14)") compiler will keep else part.*

```
f(42);    // T = int → std::is_integral<int>::value == true
// Compiler keeps only the "integral" branch

f(3.14);  // T = double → std::is_integral<double>::value == false
// Compiler keeps only the "not integral" branch
```

**** Think f(42)=print(42)*

Generated machine code for print(42) contains ***no trace*** *of the "not integral" branch.*
Generated machine code for print(3.14) contains ***no trace*** *of the "integral" branch.*

The compiler erases the irrelevant branch at compile time.

Constexpr- *is a keyword in c++. That tells to evaluate the value in compile time.*

See this from ***cppinsights.io*** *comparing with and without "constexpr"*

```cpp
#include <iostream>
#include <type_traits>
using namespace std;

template <typename T>
void print(const T& value) {
    if(std::is_integral<T>::value) {
        cout << value << " is an integral type\n";
    } else {
        cout << value << " is NOT an integral type\n";
    }
}

int main() {
    print(42);     // integral
    print(3.14);   // not integral
    cout<<"mmm";
}
```

```cpp
template<>
void print<int>(const int & value)
{
  if(std::integral_constant<bool, true>::value) {
    std::operator<<(std::cout.operator<<(value), " is an integral type\n");
  } else {
    std::operator<<(std::cout.operator<<(value), " is NOT an integral type\n"
  }

}
#endif


/* First instantiated from: insights.cpp:16 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
void print<double>(const double & value)
{
  if(std::integral_constant<bool, false>::value) {
    std::operator<<(std::cout.operator<<(value), " is an integral type\n");
  } else {
    std::operator<<(std::cout.operator<<(value), " is NOT an integral type\n"
  }

}
```

```cpp
#include <iostream>
#include <type_traits>
using namespace std;

template <typename T>
void print(const T& value) {
    if constexpr (std::is_integral<T>::value) {
        cout << value << " is an integral type\n";
    } else {
        cout << value << " is NOT an integral type\n";
    }
}

int main() {
    print(42);     // integral
    print(3.14);   // not integral
    cout<<"mmm";
}
```

```cpp

/* First instantiated from: insights.cpp:15 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
void print<int>(const int & value)
{
  if constexpr(true) {
    std::operator<<(std::cout.operator<<(value), " is an integral type\n");
  } else /* constexpr */ {
  }

}
#endif


/* First instantiated from: insights.cpp:16 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
void print<double>(const double & value)
{
  if constexpr(false) {
  } else /* constexpr */ {
    std::operator<<(std::cout.operator<<(value), " is NOT an integral type\n"
  }

```

---------x----------

Build own Concept/Custom concept:

Example 1:

```
9   template<typename T>
10  concept myint=is_integral_v<T>;
11
12  template<typename T>
13  requires myint<T>
14  T add(T a,T b){
15      return a+b;
16  }
17
18  int main(){
19      int a=12;
20      int b=12;
21      auto rs=add(a,b);
22      cout<<rs;
23  }
```

Line- 9,10: Custom concept. Here we are setting function parameter have to integral type.

Line-13: function checks whether our parameter satisfy the concept. If one parameter is float it will give compiler error.

For int value the type trait (is_integral_v) is return true and concept is satisfied. And function template is gonna execute.

For double, float value we can use "is_floating_point<T>".

Diff way to use concepts:

```
12  template<myint T>
13  T add(T a,T b){
14      return a+b;
15  }
16
```

```
12  template<typename T>
13  requires myint<T>
14  T add(T a,T b){
15      return a+b;
16  }
```

```
template<typename T>
T add(T a,T b)requires myint<T>{
    return a+b;
}
```

```
myint auto add(myint auto a,myint auto b){
    return a+b;
}
```

Example 2:

```
 9   template<typename T>
10   concept mymulti=requires(T a, T b){
11     a*b; // check if a,b is multiplyable
12   };
13
14   template<typename T>
15   requires mymulti<T>
16   T multi(T a, T b){
17     return a*b;
18   }
19
20   int main(){
21     int a=12;
22     int b=12;
23     auto rs=multi(a,b);
24     cout<<rs;
25   }
```

Line-10,11: requires 2 parameter which are multipliable. This will not give multiply of "a" ans "b" If pass (char) concepts will not satisfy. Error

Example 3: *If we want "a" will be int and "b" will be double. See below. Use diff type name.*

```
11   template<typename T, typename U>
12   concept mymulti=requires(T a, U b){
13     a*b; // check if a,b is multiplyable
14   };
15
16   template<typename T, typename U>
17   requires mymulti<T, U>
18   // requires myint<T>;
19   T multi(T a, U b){
20     return a*b;
21   }
22
23   int main(){
24     int a=12;
25     double b=12.6;
26     auto rs=multi(a,b);
27     cout<<rs;
28   }
```

----------x----------

Deep dig into 'Requires' :

you can only put valid expressions involving the types/objects. requires doesn't test boolean conditions — it just checks "is this expression well-formed?" check given expression is vakid or not.

Inside requires { ... }, each line is an expression requirement.

It only checks whether the expression is valid (well-formed) — not whether its value is true or false.

- Use `requires { expr; }` → when you just want to check if `expr` *compiles*.
- Use `concept = condition;` → when you want to enforce a *true/false property*.

```cpp
template<typename T, typename U>
concept siz=requires(T a, U b){
  sizeof(T)<=4;
  sizeof(U)<=4;
};
template<typename T, typename U>
requires siz<T, U>
T add(T a, U b){
  return a+b;
}

int main(){
  int a=12;
  double b=12.6;
  auto rs=add(a,b);
  cout<<rs;
}
```

Here "sizeof(T)<=4" is simple requirement. Only check syntax.

Although b is double. It should give compiler error cause concepts does not meet(double size 8byte). It gives ans. Cause:

sizeof(T) <= 4 is always a valid expression (it produces a true value).

The compiler says "requirement satisfied" regardless of whether the condition is true or false.

It doesn't enforce the size check.

```cpp
template<typename T, typename U>
concept siz=(sizeof(T)<=4 && sizeof(U)<=4);

template<typename T, typename U>
requires siz<T, U>
T add(T a, U b){
  return a+b;
}

int main(){
  int a=12;
  double b=12.6;
  auto rs=add(a,b);
  cout<<rs;
}
```

This is correct way to set sizeof(). This will check conditions are true or false. And this will give error cause 'b' size is 8 byte. We set <=4.

Or we can use nested requirement(requires inside requires). See below(No error)

```cpp
template<typename T, typename U>
concept siz=requires(T a, U b){
  requires sizeof(T)<=4;
  requires sizeof(U)<=8;
};

template<typename T, typename U>
requires siz<T, U>
T add(T a, U b){
  return a+b;
}

int main(){
  int a=12;
  double b=12.6;
  auto rs=add(a,b);
  cout<<rs;
}
```

*Example *: Using Logical operator...*

```cpp
template<typename T>
requires is_integral_v<T> || is_floating_point_v<T>
T add (T a, T b){
  return a+b;
}

int main(){
  double a=12;
  double b=12.6;
  auto rs=add(a,b);
  cout<<rs;
}
```

*Example **:*

```cpp
template<typename T>
concept siz=requires(T a){
  requires sizeof(T)<=8;
};

template<typename T>
requires is_integral_v<T> ||siz<T>
T add (T a, T b){
  return a+b;
}

int main(){
  double a=12;
  double b=12.6;
  auto rs=add(a,b);
  cout<<rs;
}
```

Exit(): *"exit(1)" terminate the whole program. So if this is also used in used in a function, it stop the entire program.*

`exit(1)` = "Stop the program right now — something went wrong."

`exit(0)` = "Stop the program — everything went fine."

- `exit()` doesn't "return" like a normal function.
- It calls the **system's termination process**:
  - flushes all output buffers,
  - closes all open files,
  - releases system resources,
  - and then ends the entire program.

```
exit(0);
```

it means:

➡ **Immediately terminate the entire program**, just like `exit(1)`, but the difference is in the **status code** it sends to the operating system.

exit(0) → "Program ended successfully."
exit(1) → "Program ended due to an error." // abnormal termination of the program

----------x----------

**** If we pass an array to a function is pass its reference through pointer. Not a copy.*

OOP

Class: *Class is a mechanism to build our own type to use them like we have been using built in type(int, double). Its like a blueprint to create object.*

Object: *An object is a real instance(copy) of that class. Like a actual car built from blueprint.*

```
8   class mycls{
9     public:
10    int a;
11    double b;
12    int area(){
13      return a*b;
14    }
15  };
16
17  int main(){
18    mycls ob1;
19    ob1.a=12;
20    ob1.b=12.3;
21    cout<<ob1.a<<" "<<ob1.area();
22  }
```

** Public means after "public" all parameter are accessible outside of the class.*

** If public is not defined. Then in general all parameter are indicated to private. Means can't access outside of the class. (Members of class are private by default).*

** Public, private, protected are called access specifier.*

** if public is not defined on above example. All member will be private. Line 19,20, 21 can't write. Compiler will give error.*

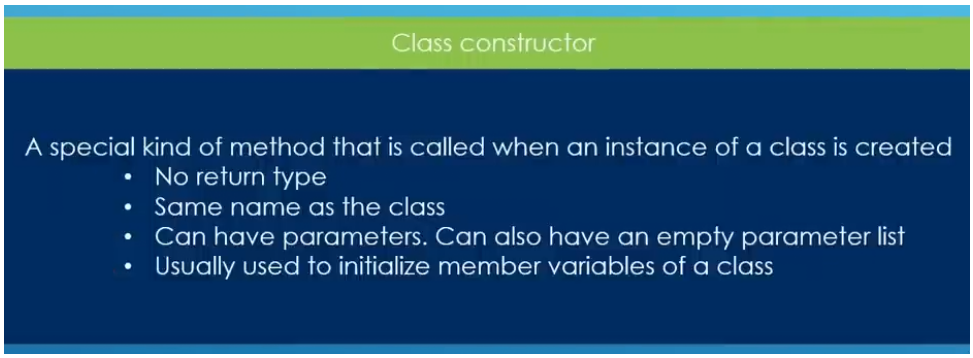
** Private members can be accessible from inside.*

** Objects(ob1) are run time data.*

** Member variable should be set to private.*

- Class member variables can either be raw stack variables or pointers
- Members can't be references
- Classes have functions (methods) that let them do things
- Class methods have access to the member variables, regardless of whether they are public or private
- Private members of classes (variables and functions) aren't accessible from the outside of the class definition

Constructor:



Constructor are 2 type.

1. Default constructor. //Method have no parameter line: 13

2. Parameterized constructor. //Have parameter Line: 17

```
8   class mycls{
9     private:
10      int a;
11      double b;
12    public:
13      mycls(){
14        a=1;
15        b=12;
16      }
17      mycls(int x, double y){
18        a=x;
19        b=y;
20      }
21      double area(){
22        return a*b;
23      }
24  };
```

** This is how private members can be accessible inside class.*

** If an object is declare default constructor will be called. Line: 13*

** if not defined(default constructor), compiler will automatically generate default empty constructor.*

Or inside main

"mycls ob1(12,22);" parameterized constructor will be called. Line: 17. default will not be called.

** If compiler sees any constructor, its not gonna generate default constructor.*

** We can declare default constructor 2 ways; on line:13 & 16*

```
9     private:
10      int a;
11      double b;
12    public:
13      mycls(){
14
15      }
16      mycls()= default;
```

Setter & Getter:

Private members are not accessible from outside. Both are methods to modify or read member variable of a class.

----------x----------