Strings

- Array of chars(char * or char[])
- Easy to corrupt (overwrite '\0')
- must explicitly allocate memory

In C++

```
"import <string>;"
  std::string -> type
  - it grows as needed
  - safer to manipulate

// C++
  string s = "Hello";
  s = "Hi";
```

Strings Operations

```
- Equality (s1 == s2, s1 != s2)
- Comparison: s1 <= s2
- Length (s.length()) //O(1)
- individual chars (s[0], s[1])
- concat: s3 = s1 + s2
          s3 += s4
- cin >> s; //!!! read a word
              - ignores leading white space
              - stops at next white space
- getline(cin(stream), s(string))  // read in "a line"
```

Example in VSCODE

Streams are **abstraction**

Abstraction:

- Wrap an interface of getting and putting items to keyboard and screen

Files (file streams is an abstraction)

- Read/write from/to a file instead cin/cout

```
std::ifstream //a file stream for reading
```

```
std::ofstream // a file stream for writing

Ex: in C
#include <stdio.h>
int main () {
  char s[256]; // Hoping no words is larger than 255 chars
  FILE *f = fopen("file.txt", "r");
  while (1) {
    fscanf(f, "%255s", s);
  }
}

in C++:
import <iostream>;
import <fstream>;
using namespace std;

int main () {
  ifstream f{"file.txt"}; // Declaring and initializing opens the file
  string s; // declare string variable; no concerns about size of the words
  while (f >> s) { //f getting from s
    cout << s << endl;
  }
} // not closing the file because of "f". file is closed when it is out of scope
```

Command Line Arguments:

```
./program abc 123

//To access these args:
int main(int argc, char* argv[]) {
  // argc: # of arrays given to the program - including program name
  // argb -> array of char *s, i.e - C style Strings
  // argv[c] - name of program
  // argv[1] - 1st arg
  // argv[2] - 2nd arg
  ..
  // argv[argc] - Null Pointer
}
```

Stack

argv: _ _ _ _

Index 0: ./program\0

Index 1: a b c /0

Index 2: 1 2 3 /0

Index 3: NULL

!!! **WRONG**

```
if (argv[1] == "abc") // Comparing pointer locations instead of String contents!!!!
```

Advice: Convert the args to **std::String**

Before use:

```
int main(int argc, char **argv) { // char**argv Equivalent to *argv
  for (int i = 0; i < argc; ++i) {
    String arg = argv[c];
    if (arg == "abc") {} //WORKS!
  }
}
```

Ex: Add all integers given as args on the cmd Line:

```
int main (int argc, char *argv[]) {
  int sum = 0;
  for (int i = 1; i < argc; ++c) {
    string arg = argv[i];
    int n;
    if (istringstream iss{arg}; iss >> n) sum += n;
  }
  cout << sum << endl;
}
```

Default Function Parameters

```
void printSuiteFile (string fileName = "suite.txt") {
  ifstream file {fileName};
  string line;
  while (getline(file, line)) cat << line << endl;
}

// printSuiteFile("abc.txt") -> Read from abc txt
// printSuiteFile() -> Read from file - suite.txt
// Notes: optional parameters must be last
```

Who's responsibility is it to provide the default argument? Who writes suite.txt into the stack frame of arguments?

- The caller sets up the argument, not the function itself
- printSuiteFile() is replaced with printSuiteFile("suite.txt") where it is found

void f(int x, string s = "Hello")

Int main() { f(5); }

stack: _ _ main_ (see WeChat visualization)

The defaulted function (f)

- does not know what will be passed in
- It **cannot** detect the difference between uninitialized memory and other strings
- Caller of f must provide **all arguments** to f in advance

Note: Default parameters are part of the **declaration**, not necessary definition

```
in Header (interface) file:
void f(int x; string s = "Hello");

// in .c file
void f(int x, string s) {}
```

Default parameters are given in the **interface (header file in this case)**, **not** the implementation (.c / .cc file)

Overloading

- if I want a function to negate ints and bool
    - in C:

```
int negInt(int x) {return -x;}
bool negBool(bool b) {return !b;}
```

- in C++:

```
int neg(int x) {return -x;}
bool neg(bool b) {return !b;}
// neg(5) -> -5
// neg(true) -> false
```

- Compiler chooses which **overload** of the function to use based on the **#** and the **types of the args** at compile line.
  - seen this in the case of operators: ==, <<, >>, +
  - We cannot overload based on the **return type alone**
  - can't have same args in overloading functions. s.t. f(int x); f(int x)

Structs

- Structs also exist in C++, just as they existed in
  - Ex: Linked list node:

```
struct Node {
  int data;
  Node* next;
}; // dont forget ;
```

In C: this would be Struct Node*

in C++: **omit** the struct

Recall: why is this invalid?

```
struct Node {
  int data;
  Node next;
};
```

- What is the size of the struct?
  - we need to solve the following:
    - **sizeof(Node) = sizeof(int) + sizeof(Node)** X=4+X

- The previous (*Node) works because ptrs are if fixed size;
  - 8 bytes no matter what you point at

Constant

```
const int max_grade = 100;
// Any attempt to modify this variable wont compile

const Node n {5, nullptr};
// n's data cannot be changed and n's next field cannot be changed
```

In C:

```
// The null ptr is inficated via NULL or 0
// wont do this in C++!!!
```

- in C++:
  - Typically in C - there is the following line imported via a library
    - #defines NULL 0 -> replaces all instances of NULL with 0
    - consider following scenario:

```
void f(int x);
void f(Node* p);

1. calls f(NULL) - calls int version of f instead of pounter version


// nullptr is a special type that can be automatically converted to any other ptr
type
2. f(nullptr) -> calls ptr version of f
```

Parameter Passings:

```
void inc(int x) {++x;}

int main() {
  int x = 5;
  inc(x);
  cout << x <<endl; // output 5 because we made a copy of 5 and call the function (AKA
pass by value semantics)
}
```

Pass-by-value semantics: Copy arguments into the function instead of using orginal.

```
void inc(int *p) {++*p;}

int main() {
  int x=5;
  inc(&x); // pass by reference
  cout << x << endl;
}
```

- If we want to mutate an argument and change the value outside the function
    - pass a pointer

C++ has another pointer-like type: **Reference**

References:

```
int x = 5;
int& z=x; // any changes to z will be reflected in x (lvalue)
// z is an "alias" to the value of x
// anytime you see z, simply imagine replacing it with x

++z;
cout << z <<" "<<x<<endl;
// z=x=6
```

- References are somewhat like **const ptrs** with automatic dereference.
- We cannot change what a reference is aliasing
- When does & mean reference vs when it is address-of?
    - & in a **type** means references, **in an expression it means address-of**
    - f(int& x) // int&: reference

    - Int* y = &x; // expression: address-of
- Things you **cannot** do with references:
    - Leave them uninitized, must alias a variable at all times. (**MUST BE INITIALIZED**)

```
int &z; //INVALID: WONT COMPILE
```

- Lvalue references must always be initialized with lvalues
- Lvalue: a value you may take the address-of
    - Rvalue below:

```
int& z = 5; //cannot take the address of 5, it is not an lvalue - it is an rvalue
```

    - lvalues have a "name" - which tells you where in memory the variable is stored!
        - rvalue again:

```
int& z = x + y; // rvalue - tempaorary variable
int& z = f(); // wont compile
```

- We cannot **create a pointer** to a **reference**:
    - read from right to left
        - int&*x; // x *wont* compile, ptr to a reference
        - Int*&x; // reference to a ptr - this is *allowed*
- Cannot create a **refernce** to a **reference**

```
int&& x; //This is not a reference to a reference;
         // this is a rvalue reference
```

- Cannot create an array of references

```
int&arr[3] = {x,x,x}; // X wont compile
// Reference Assignment: Once a reference is initialized, it cannot be reassigned
to refer to a different object. If you had an array of references, you wouldn't
be able to change what they refer to after initialization.
```

    - what are references useful for?
        - Most commonly: Use as **args** to a function

```
// we saw
inc(int x) and inc(int* p) {++*p;}
```

- with references:

```
void inc(int& x) {++x;} // automatically dereference
x = 5;
inc(x);
cout<<x<<endl; //6
```

- How does int x; cin>>x work?

```
cin >> x // actually just a fn call
// how it is implemented in stardard library
istream& operator>>(istream& in, int& x) {
  // read from in into X
  return in;
}
```

- Is there a good reason why we take in and return istream& instrad of an istream?
  - **pass by value** can be an expensive operation
  - copying memory from caller's stack frame into called stack frame
  - ex:

```
struct ReallyBig {...};
void f(ReallyBig rb) {...};// no changes reflected, pass by copy (copy all
mempory in rb -> expensive)
void g(ReallyBig& rb) {...} // Changes are reflected, not expensive, this is
fast- 8 bytes copied
void h(const ReallyBig& rb) {...} //FAST; rb wont be changed; should be a good
choice
```

- When designing a function with argiments
  - **pass-by-const-reference** should be first choice
  - use pass-by-ref only if you **need** to changes reflected outside function call

```
pass-by-value: 2 scenarios;
1) Arg is small-pointer size of smaller (< 8bytes)
2) void a(const ReallyBig& rb) {
  ReallyBig temp {rb}; // just use pass-by-value instead, copies for you ???
}
```

```
in istream& operator>>(istream& in, int&x);
// pass cin via reference to avoid expenses of copying
// (streams have copying disabled - copying does not compile)
void f(int&x); // can only pass by lvalue
void g(const int&y); // can pass both lvalue and rvalue: g(z), g(5) workss
```

- **const** - lvalue references like in g(ex above) can bind to rvalues (tempraries) because we know they won't be changed
    - Compiler create a tempory location and store the temporary (ex: g(5) where 5 is the temporary memory) inside

Dynamic Memory

- In C:

```
int *p = malloc(n * sizeof(int)); //dynamic array of int
... //maybe resize...
free(p); // give memory back to OS
```

- In C++: use new/delete

```
Node* np = new Node {5, nullptr};
...
delete np; // give back memory to OS; follows np's memory address and heap and delete
it, but in stack, it changes nothing
```

- Important!
    - All local variables rewrite space on the stack
    - use new to acquire memory from the heap
    - calling delete returns memory to OS to be reused,
    - does not affect value of pointer at all
    - Stack allocated vars; // automatically deallocated when they leav scope
    - Heap allocated memory exists until delete is called
    - A memory leak occurs when memory that is allocated with new is new deleted.
- C++: dynamic arrays;

```
int* arr = new int[10]; // array of 10 ints on heap;
                        // arr points to first element in the arr
...
delete[] arr; // frees a dynamically allocated array
```

- Always pair **new + delete** new[] with delete[]
- Do not mix and match or else undefined behaviour

Lecture 5

We have seen for args: pass-by-value, pass-by-ptr, pass-by-ref

Which is the best option for returns? //

```
Node getMeANode(int n) {
  return Node {n, nullptr}; // return-by-value
}
```

This works - make a copy from getMeANode's stack frame into the caller's frame: Too slow?

not working version:

```
Node* getNodeptr(int n) {
  Node x {n, nullptr};
  return &x; // wrong: it returns the pointer into the stack frame; returing a dangling
pointer;
             // x will be cleaned up when fn returns, pointing at memory we no longer own.
}
```

working version:

```
Node* getNodeptr(int n) {
  Node* p = new Node {n, nullptr};
  return p;
}
```

This is fast and works **BUT** user must remeber to **delete** heap allocated memory, or else **memory leak**

Return by Ref:

```
Node& getNodeptr(int n) {
  Node x{n, nullptr};
  return x; // still not working because acess to the alias of the node, we still lose x
            // dangling ref - same problem as ptr example
}
```

- Returning by reference is rare.
- Returning a reference to stack allocated variables will cause undefined behaviour.
- Only use return-by-ref if returning a ref to a non-local variables. Eg:

```
istream& operator>>(istream& in, int& x) {
  ...
  return in; // works because in is not local variable
}
```

- Advice: Usually, **return-by-value**: easy to use, not as slow as it may seem (more semantics)

Operator Overloading

- Defined the meaning of operators acting on our own types

```
struct vec {
  int x, y;
};

vec operator+(const vec& v1, const vec& v2) {// const reference of v1; aliases v1
  vec r{v1.x+v2.x, v1.y+v2.y};
  return r;
}

vec operator*(int k, const vec& v) {
  return {k*v.x, k*v.y}; // vec could go here to explicityly construct a vec, obied,
but assumed by return type
}

vec z = w * 2; //This doesnt mach our first overload - order of args is wrong

vec operator*(const vec& x, int k) {
  return k*v; // int * vec from the previous function. calls the other operator*
}
```

Example: Overloading << and >>

```
struct grade { // support: grade g; cin>>g; cout<<g;
  int n;
};

istream& operator>>(istream& in, grade& g) {
  in>>g.n;
  if (g.n < 0) g.n=0;
  if (g.n > 100) g.n=100;
  return in;
}

ifstream file {"file.txt"};
grade g;
file>>g; // works

ostream& operator<<(ostream& out, const grade& g) {
  return out<<g.n<<'%';
} // generally, no need for operator<<, it would force users to print newlines
```

Separate Compilation

Point: Speed up compilation times, by only recompiling what we need to

Recall:

- Interface files (think .h files): provide declarations
- Inplementation files (think .c): provide definitions

```
// interface (vec.cc)
export module vec; // indicates an interface file
export struct vec z { // Anything marked export clients may use
  int x,y;
}
export vec operator+(const vec& v1, const vec& v2);

// Client code (main.cc)
import vec;
int main() {
  vec v{1,2};
  vec w = v+v;
}
```

```
// Implementation file: (vec-implementation.cc)
module vec;
vec operator+(const vec& v1, const vec& v2) {
  return {v1.x+v2.x, v1.y+v2.y};
}
```

Recall:

- we can repeatedly declare structs/functions, but we may only define once.
- Interface files: export module x;
- implementation files: module x;

To create an object file (.o file):

```
g++20m vec.cc -c // -c means compile only
                 // creates vec.o
```

with c++20 modules, object files must be created in **dependency** order

- First compile **interface files**, then **implmentation** / client files!!! (ORDER BELOW)
  - vec.cc
  - vec-impl.cc
  - main.cc

```
g++20m vec.o vec.impl.o main.o -o program //links object files
```

Building dependencies with Make in progress.

```
g++20m *.cc -c // fail on some which are out of order
// can write g++20m *.cc without -c
g++20 *.cc -c // create remaining .o files
g++20m *.o -o program // linking
```

CLASSES!!

- we can put functinns inside our structs
  - // Student.cc

```
export module Student;
export Struct Student {
  int assns, mt, final;

  float grade();
}
```

- // Student-impl.cc

```
float Student::grade() { // Student:: means "scope resolution operator"
  return 0.4*assns + 0.2*mt + 0.4*final;
}

// C::f - inside of the Class C - define, member function f
```

```
Student s{60, 70, 80}; // s - is an object, instantiation of a class
s.grade(); // returns 60*0.4+70*0.2+80*0.4

// Student - this is a class - struct with functions defined inside it.
// grade() function calls member functions / method
```

- Inside Student::grade we use variables assns, mt, final
- These variables are bound to the object the method is called on

```
s.grade() // assns is s.assns
          // mt is s.mt


// When we call s.grade(), there is a hidden parameter that is used to access
assns, mt, final
this -> a pointer to the object the methods was called.

s.grade() // inside grade, this == &s



// we could write:
export struct Student {
  int assns, mt, final;
  // method
  float grade() {
```

```
// provide mthod defitions inside of class - but generally best to separate into
interface/implementation file
    return this->assns * 0.4+ this->mt * 0.2 + this->final * 0.4;
  }
};
```

Initializing objects

```
Student S {60, 70, 80};
- To better control initialization, write a constructor (ctor) will get called whenever a
Student object is created.

Struct Student {
  Student(int assns, int mt, int final) {
    // name of the method is the name of the class
    this->assns = assns;
    this->mt = mt;
    this->final = final;
  }
};

// using this-> to differ between arg and field

Student s{60, 70, 80};
Student *p = new Student{60, 70, 80};
//Invokes a ctor call - passed as args to ctor method.
// If we dont write a ctor - we get C-style field-be-field initialization

Student S = Student{60, 70, 80};
Same as Student S{60, 70, 80};
// Benefit of ctors: write our own logic - use them as functions. eg: default params:
Sturct Student { // default constuctor
  Student(int assns=0, int mt=0, int final=0) {
    this->assns = assns;
    this->mt = mt;
    this->final = final;
  }
};

Student s(60, 70, 80);
Student newkid; // newkid sets to 0
Student toby{40, 50}; // final=0 for default
```

```
Student newKid; // calls our 0 -arg ctor
A ctor that may be called with 0 args;
This is a """default ctor""". If you do not write a default ctor - compiler provides one.
// As soon as you write a ctor - compiler procided default ctor goes away
```

Compiler provides default ctor:

- Primitive types - leave them uninitialized //原始type
- Classes/ Structs - calls their default ctor.

1. **Primitive Types**:
   - Examples of primitive types include `int`, `float`, `double`, `char`, etc.
   - If you create a variable of a primitive type without initializing it, the value of that variable is left uninitialized in the case of local (stack) variables. This means the value is essentially whatever garbage data was in that memory location. For global or static variables, they are zero-initialized by default.

     ```
     cppCopy code
     int a;    // uninitialized value for local variable
     static int b; // zero-initialized
     ```

2. **Classes/Structs**:
   - If you define a class or struct without a default constructor, the compiler will implicitly define a default constructor for you, as long as there isn't any user-defined constructor present.
   - The compiler-provided default constructor will do the following:
     - It will call the default constructors for any member objects.
     - It will leave primitive type members uninitialized (just like the rule for standalone primitive types).

     For instance, consider the following example:

```
cppCopy code
struct Point {
    int x, y;
};

class Circle {
    Point center;
    double radius;
};
```

The default constructor provided by the compiler will be called. This constructor will call the default constructor of `Point` (which in this case does nothing and leaves `x` and `y` uninitialized) and leaves `radius` uninitialized as it is a primitive type.

```
Struct vec {
  vec(int x, int y)  {
    this->x = x;
    this->y = y;
  }
};
// vec v; // should call default ctor but none exist - doesnot compile.

Struct Basis {
  Vec v1, v2;
};
// Basis b; what happened?
// vec is the default ctor. (calls default constructor)
// wont compile, we cannot default construct v1 and v2 since it needs two parameters

Struct Basis {
  vec v1, v2;
  Basis() {
    v1 = vec{0, 1};
    v2 = vec(1, 0);
  }
}; // doesnt work because v1 and v2 must have values initialized by the time the function starts to run;
//v1 and v2 are being assigned values inside the constructor body. This means they are default-constructed first and then assigned new values inside the constructor. If the vec type doesn't have a default constructor, this would result in a compilation error.


Instead, you should use an initializer list to directly initialize v1 and v2:
```

```
Struct Basis {
  vec v1, v2;
  Basis() : v1{0, 1}, v2(1, 0) {
  }
};
// Now, v1 and v2 are directly initialized with the provided values when an instance of Basis is created. This is more efficient and avoids potential issues if vec does not have a default constructor.
```