WE ALL HAVE STRUGGLED WITH DYNAMIC MEMORY ALLOCATION IN C

FLASHBACK TO C

DYNAMIC MEMORY ALLOCATION

MEMORY ALLOCATION

A C PROGRAM RUNS USING SOME MEMORY FROM YOUR MACHINE - A SPECIFIC PORTION OF THIS MEMORY IS ALLOCATED TO YOUR PROGRAM

EACH LOCATION IN MEMORY HAS AN ADDRESS

WE REFER TO THESE ADDRESSES IMPLICITLY USING VARIABLES AND MORE EXPLICITLY WHEN WE USE POINTERS

SO FAR WE'VE POINTED TO VARIABLES WHICH HAVE BEEN ASSIGNED TO SOME MEMORY LOCATION BY THE COMPUTER

FLASHBACK TO C

MEMORY ALLOCATION

WHAT IF WE SET UP A POINTER AND THEN WANT TO **ALLOCATE SOME MEMORY** FOR IT?

WHAT IF WE SET UP AN ARRAY AND WANT TO SET UP SOME MEMORY FOR IT TO STORE **N** INTEGERS? HERE IS **N** IS SOME VALUE WHICH THE USER SPECIFIES AT RUNTIME

FLASHBACK TO C

MEMORY ALLOCATION

```
void *malloc(int num);
```

THIS A SPECIAL FUNCTION PROVIDED BY THE STDLIB.H LIBRARY IN C TO ALLOW DEVELOPERS TO MANAGE MEMORY

IF YOU, AS A DEVELOPER ALLOCATE MEMORY, THEN IT'S YOUR RESPONSIBILITY TO FREE IT AS WELL!

```
void free(void *address);
```

FLASHBACK TO C

MEMORY ALLOCATION

```
void *malloc(int num);
```

RETURN VALUE

A **VOID*** POINTER INDICATES THAT THE POINTER CAN BE TO **ANY DATA TYPE**

THIS CAN BE **CAST** TO THE **TYPE OF POINTER** WE'RE ALLOCATING MEMORY FOR

NUMBER OF BYTES

THIS SPECIFIES **HOW MUCH MEMORY**, IN BYTES, WE WANT TO ALLOCATE

FLASHBACK TO C

MEMORY ALLOCATION

```
int* int_ptr = (int *)malloc(sizeof(int));
```

HERE WE ALLOCATE MEMORY FOR AN INTEGER POINTER, CAST THE RETURN VALUE TO AN INT*

SIZEOF CAN BE USED WITH PRIMITIVES, STRUCTS AND CONSTANTS TO FIGURE OUT HOW MUCH SPACE EACH OF THEM OCCUPY

FLASHBACK TO C

MEMORY ALLOCATION

SIZEOF RETURNS AN INTEGER VALUE AND CAN BE USED IN ARITHMETIC AND OTHER OPERATIONS

```
sizeof(int)
```
RETURNS 4

```
sizeof(char)
```
RETURNS 1

```
sizeof(char*)
```
RETURNS 4

```
struct Point3D {
   int x;
   int y;
   int z;
}
```

```
sizeof(struct Point3D)
```
RETURNS 4 + 4 + 4 = 12

FLASHBACK TO C

WHAT YOU ALLOCATE
YOU MUST FREE!

FLASHBACK TO C

WHAT YOU ALLOCATE YOU MUST FREE!

THE ONUS OF **CLEANING UP MEMORY** LIES WITH THE DEVELOPER

ANY MEMORY ALLOCATED USING **MALLOC** HAS TO BE CLEANED UP BY USING **FREE**

```
void free(void *address);
```

IF YOU ONLY ALLOCATE AND DO NOT FREE MEMORY ANY LONG RUNNING PROGRAM WILL **RUN OUT OF SPACE**!

AS YOUR PROGRAM GETS MORE COMPLEX - FREEING MEMORY BECOMES PAINFUL AND COMPLICATED

FLASHBACK TO C

WHAT YOU ALLOCATE YOU MUST FREE!

```
void free(void *address);
```

FREE RETURNS NOTHING, SIMPLY FREES UP THAT PORTION OF THE MEMORY FOR USE BY OTHER PORTIONS OF THE PROGRAM

IT TAKES IN AN **VOID*** WHICH MEANS YOU CAN PASS IN A POINTER TO ANY DATA TYPE

FLASHBACK TO C

FREEING MEMORY

```
int* int_ptr = (int *)malloc(sizeof(int));
int a = 34;
…
free(int_ptr);
free(a);
```

THIS FREES THE MEMORY SPACE THAT INT_PTR USED

YOU **CANNOT FREE** THE INTEGER A, YOU HAVE NOT ALLOCATED MEMORY FOR IT.

THE CPU ALLOCATES MEMORY FOR A AND THE CPU WILL FREE IT

FLASHBACK TO C

STACK MEMORY VS HEAP MEMORY

FLASHBACK TO C

STACK MEMORY VS HEAP MEMORY

IN C THERE ARE 2 KINDS OF MEMORY AVAILABLE TO USE

STACK MEMORY

THE CPU MANAGES THE STACK MEMORY - THE CPU IS RESPONSIBLE FOR ALLOCATING SPACE IN IT AND FREEING UP SPACE AS WELL

DEVELOPERS DO NOT ALLOCATE OR DE-ALLOCATE SPACE ON THE STACK

HEAP MEMORY

THIS PORTION OF MEMORY IS MANAGED BY THE DEVELOPER WRITING CODE IN C

THE DEVELOPER IS RESPONSIBLE FOR ALLOCATING AND FREEING MEMORY ON THE HEAP

FLASHBACK TO C

STACK MEMORY

MEMORY ALLOCATED FOR VARIABLES WHICH WE JUST DECLARE AND USE IS ON THE STACK

```
int a;
char c;
float f;
struct point3D;
```

THE MEMORY FOR ALL VARIABLES DECLARED IN THIS FASHION ARE PUT ON THE STACK MANAGED BY THE CPU

IT'S CALLED A STACK BECAUSE IT BEHAVES LIKE A STACK!

FLASHBACK TO C

STACK MEMORY

main() MAIN IS THE ENTRY POINT AND HAS VARIABLES DECLARED IN IT

func1() SOMEWHERE IN MAIN WE CALL FUNC1

func2() FUNC1 CALLS FUNC2

func3() FUNC2 CALLS FUNC3

FLASHBACK TO C

STACK MEMORY

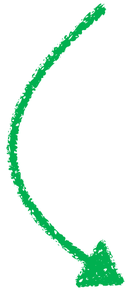
func1() SOMEWHERE IN MAIN WE CALL FUNC1

func2() FUNC1 CALLS FUNC2

func3() FUNC2 CALLS FUNC3

FLASHBACK TO C

STACK MEMORY

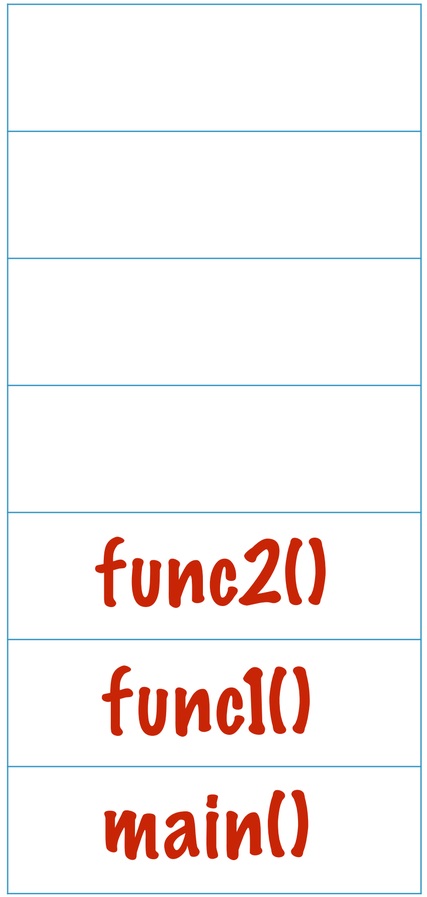


func2() FUNC1 CALLS FUNC2

func3() FUNC2 CALLS FUNC3

FLASHBACK TO C

STACK MEMORY

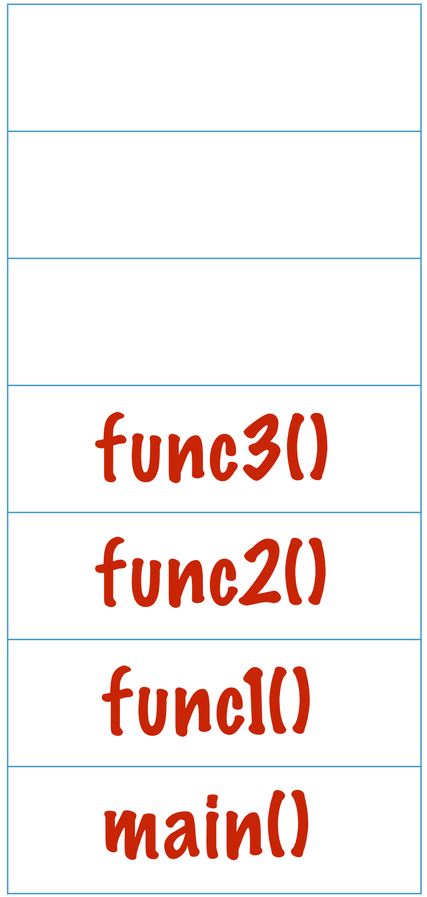


func3()

FUNC2 CALLS FUNC3

FLASHBACK TO C

STACK MEMORY



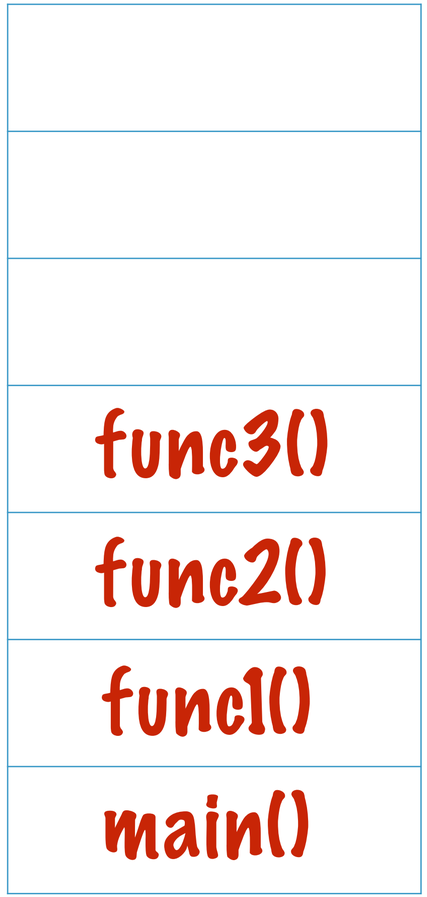
THE VARIABLES ASSOCIATED WITH EACH FUNCTION IS ADDED TO THE STACK MEMORY

FUNCTIONS CALLED LATER ARE ADDED TO THE TOP

AS WE RETURN FROM FUNCTIONS (UNWIND) THE VARIABLES ASSOCIATED WITH THE FUNCTION ARE REMOVED ENTIRELY

FLASHBACK TO C

STACK MEMORY



SAY WE RETURN FROM FUNC3

VARIABLES DECLARED IN FUNC3 ARE
NO LONGER AVAILABLE FOR US TO USE!

FLASHBACK TO C

STACK MEMORY

THE STACK GROWS AND SHRINKS AS FUNCTIONS PUSH AND POP LOCAL VARIABLES - NEWER VARIABLES GO TO THE TOP OF THE STACK

VARIABLES IN THE STACK ARE MANAGED BY C AUTOMATICALLY

STACK VARIABLES ONLY EXIST SO LONG AS THE FUNCTION WHICH CREATED THEM IS BEING EXECUTED

FLASHBACK TO C

STACK MEMORY VS HEAP MEMORY

IN C THERE ARE 2 KINDS OF MEMORY AVAILABLE TO USE

STACK MEMORY

THE CPU MANAGES THE STACK MEMORY - THE CPU IS RESPONSIBLE FOR ALLOCATING SPACE IN IT AND FREEING UP SPACE AS WELL

DEVELOPERS DO NOT ALLOCATE OR DE-ALLOCATE SPACE ON THE STACK

HEAP MEMORY

THIS PORTION OF MEMORY IS MANAGED BY THE DEVELOPER WRITING CODE IN C

THE DEVELOPER IS RESPONSIBLE FOR ALLOCATING AND FREEING MEMORY ON THE HEAP

FLASHBACK TO C

HEAP MEMORY

THE HEAP REGION OF YOUR COMPUTER'S MEMORY IS NOT MANAGED AUTOMATICALLY

MEMORY FOR POINTERS IS ALLOCATED IN THE HEAP REGION

THE DEVELOPER IS RESPONSIBLE FOR MANAGING MEMORY USING FUNCTIONS LIKE `malloc()` AND `free()`

IF YOU FAIL TO DEALLOCATE MEMORY USED ON A HEAP IT RESULTS IN A MEMORY LEAK

FLASHBACK TO C

STACK MEMORY VS HEAP MEMORY

IN C THERE ARE 2 KINDS OF MEMORY AVAILABLE TO USE

STACK MEMORY

THE CPU MANAGES THE STACK MEMORY - THE CPU IS RESPONSIBLE FOR ALLOCATING SPACE IN IT AND FREEING UP SPACE AS WELL

DEVELOPERS DO NOT ALLOCATE OR DE-ALLOCATE SPACE ON THE STACK

HEAP MEMORY

THIS PORTION OF MEMORY IS MANAGED BY THE DEVELOPER WRITING CODE IN C

THE DEVELOPER IS RESPONSIBLE FOR ALLOCATING AND FREEING MEMORY ON THE HEAP

FLASHBACK TO C

STACK MEMORY

FAST ACCESS

SPACE MANAGED AUTOMATICALLY AND EFFICIENTLY - DOES NOT GET FRAGMENTED

VARIABLES ARE IN THE SCOPE OF THE FUNCTION DECLARED

SMALLER IN SIZE - HAS SIZE LIMITS

HEAP MEMORY

ACCESS IS SLOWER

MANAGED BY THE DEVELOPER AND CAN GET FRAGMENTED

VARIABLES CAN BE ACCESSED ANYWHERE IN THE PROGRAM

LARGER - TYPICALLY LIMITED BY THE MEMORY ON YOUR COMPUTER

FLASHBACK TO C

OTHER WAYS TO ALLOCATE MEMORY

```
void *calloc(int num, int size);
```

THIS IS A SPECIAL FUNCTION WHICH IS USEFUL TO ALLOCATE MEMORY FOR **ARRAYS**

THIS WILL ALLOCATE MEMORY FOR **NUM** ELEMENTS WHERE EACH ELEMENT IS OF **SIZE** BYTES

FLASHBACK TO C

ALLOCATING MEMORY FOR ARRAYS

SAY YOU HAVE TO ALLOCATE MEMORY FOR A 100 INTEGER ELEMENTS IN AN ARRAY

```
int* arr = (int *)malloc(100 * sizeof(int));
```

IS EQUIVALENT TO:

```
int* arr = (int *)calloc(100, sizeof(int));
```

FLASHBACK TO C

RESIZING A BLOCK OF MEMORY

IF YOU HAVE ALREADY ALLOCATED SOME MEMORY FOR AN ARRAY, LATER YOU REALIZE THAT YOU WANT TO STORE ADDITIONAL ELEMENTS

ORIGINAL ALLOCATION

```
int* arr = (int *)malloc(100 * sizeof(int));
```

REALLOCATION WITH INCREASED SIZE

```
arr = (int *)realloc(arr, 200 * sizeof(int));
```

FLASHBACK TO C

RESIZING A BLOCK OF MEMORY

```
int* arr = (int *)malloc(100 * sizeof(int));
arr = (int *)realloc(arr, 200 * sizeof(int));
```

THE RE-ALLOCATED MEMORY BLOCK WILL START AT THE SAME ADDRESS AS THE ONE ORIGINALLY ALLOCATED

IT WILL TRY TO EXTEND THE BLOCK OF MEMORY FROM THE END - WHICH MEANS ORIGINAL ELEMENTS FROM THE ARRAY WILL BE PRESERVED

FLASHBACK TO C

WHAT IF MEMORY IS NOT AVAILABLE?

EACH OF THESE FUNCTIONS TO ALLOCATE MEMORY **MALLOC()**, **CALLOC()** AND **REALLOC()** WILL RETURN A NULL POINTER IF THE MEMORY IS NOT AVAILABLE

WE ALSO JUST LEARNT ABOUT OBJECTS, CLASSES, CONSTRUCTORS AND DESTRUCTORS

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

SOME OF THESE VARIABLES AND FUNCTIONS MIGHT BE INTERESTING TO THE REST OF THE PROGRAM, AND THESE SHOULD BE PUBLIC

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

SOME OF THESE VARIABLES AND FUNCTIONS MIGHT BE INTERESTING TO THE REST OF THE PROGRAM, AND THESE SHOULD BE PUBLIC

OTHERS MIGHT BE INTERNAL PLUMBING, AND THESE SHOULD BE PRIVATE

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

SOME OF THESE VARIABLES AND FUNCTIONS MIGHT BE INTERESTING TO THE REST OF THE PROGRAM, AND THESE SHOULD BE PUBLIC

OTHERS MIGHT BE INTERNAL PLUMBING, AND THESE SHOULD BE PRIVATE

THIS GROUP OF VARIABLES & FUNCTIONS IS EFFECTIVELY A NEW, USER-DEFINED, TYPE

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

OTHERS MIGHT BE INTERNAL PLUMBING, AND THESE SHOULD BE PRIVATE

SOME OF THESE VARIABLES AND FUNCTIONS MIGHT BE INTERESTING TO THE REST OF THE PROGRAM, AND THESE SHOULD BE PUBLIC

THIS GROUP OF VARIABLES & FUNCTIONS IS EFFECTIVELY A NEW, USER-DEFINED TYPE

ANYONE CAN CREATE A VARIABLE OF THIS TYPE

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

SOME OF THESE VARIABLES AND FUNCTIONS MIGHT BE INTERESTING TO THE REST OF THE PROGRAM, AND THESE SHOULD BE PUBLIC

OTHERS MIGHT BE INTERNAL PLUMBING, AND THESE SHOULD BE PRIVATE

THIS GROUP OF VARIABLES & FUNCTIONS IS EFFECTIVELY A NEW, USER-DEFINED, TYPE

ANYONE CAN CREATE A VARIABLE OF THIS TYPE

SUCH A VARIABLE MIGHT REQUIRE SOME INITIALISATION, AND ONCE ITS DONE, SOME CLEAN-UP

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

OTHERS MIGHT BE INTERNAL PLUMBING, AND THESE SHOULD BE PRIVATE

ANYONE CAN CREATE A VARIABLE OF THIS TYPE

SOME OF THESE VARIABLES AND FUNCTIONS MIGHT BE INTERESTING TO THE REST OF THE PROGRAM, AND THESE SHOULD BE PUBLIC

THIS GROUP OF VARIABLES & FUNCTIONS IS EFFECTIVELY A NEW, USER-DEFINED, TYPE

SUCH A VARIABLE MIGHT REQUIRE SOME INITIALISATION, AND ONCE ITS DONE, SOME CLEAN-UP

AND COOLEST OF ALL, OTHER TYPES CAN "BUILD ON" THIS TYPE

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

OTHERS MIGHT BE INTERNAL PLUMBING, AND THESE SHOULD BE PRIVATE

ANYONE CAN CREATE A VARIABLE OF THIS TYPE

SOME OF THESE VARIABLES AND FUNCTIONS MIGHT BE INTERESTING TO THE REST OF THE PROGRAM, AND THESE SHOULD BE PUBLIC

THIS GROUP OF VARIABLES & FUNCTIONS IS EFFECTIVELY A NEW, USER-DEFINED, TYPE

SUCH A VARIABLE MIGHT REQUIRE SOME INITIALISATION, AND ONCE ITS DONE, SOME CLEAN-UP

AND COOLEST OF ALL, OTHER TYPES CAN "BUILD ON" THIS TYPE

C++ RECAP

"CLASSES ARE STRUCTS THAT CONTAIN FUNCTIONS"

OFTEN, SOME VARIABLES AND FUNCTIONS JUST MAKE SENSE GROUPED TOGETHER

OTHERS MIGHT BE INTERNAL PLUMBING, AND THUS SHOULD BE PRIVATE

SOME OF THESE VARIABLES AND FUNCTIONS MIGHT BE INTERESTING TO THE REST OF THE PROGRAM, AND THUS SHOULD BE PUBLIC

THIS GROUP OF VARIABLES & FUNCTIONS EFFECTIVELY A NEW, USER-DEFINED TYPE

ANYONE CAN CREATE A VARIABLE OF THIS TYPE

SUCH A VARIABLE MIGHT REQUIRE SOME INITIALISATION, AND ONCE ITS DONE, SOME CLEAN-UP

AND COOLEST OF ALL, OTHER TYPES CAN "BUILD ON" THIS TYPE

TAKEN TOGETHER, THESE IDEAS PRETTY MUCH DEFINE OBJECT ORIENTED PROGRAMMING

C++ RECAP

THIS GROUP OF VARIABLES & FUNCTIONS IS EFFECTIVELY A NEW, USER-DEFINED, TYPE

THIS USER DEFINED TYPE IS CALLED THE CLASS, AND IT CORRESPONDS EXACTLY TO A STRUCT IN C

C++ GOES FAR BEYOND C IN MAKING USER-DEFINED CLASSES FIRST CLASS TYPES, ON PAR WITH THE SYSTEM TYPES SUCH AS INT, FLOAT ETC

ANYONE CAN CREATE A VARIABLE OF THIS TYPE

C++ RECAP

THIS GROUP OF VARIABLES & FUNCTIONS IS EFFECTIVELY A NEW, USER-DEFINED, TYPE

THIS USER DEFINED TYPE IS CALLED THE CLASS, AND IT CORRESPONDS EXACTLY TO A STRUCT IN C

C++ GOES FAR BEYOND C IN MAKING USER-DEFINED CLASSES FIRST CLASS TYPES, ON PAR WITH THE SYSTEM TYPES SUCH AS INT, FLOAT ETC

ANYONE CAN CREATE A VARIABLE OF THIS TYPE

A VARIABLE OF THIS CLASS IS CALLED AN OBJECT OF (OR AN INSTANCE OF) THE CLASS

C++ RECAP

A VARIABLE OF THIS CLASS IS CALLED AN OBJECT OF (OR AN INSTANCE OF) THE CLASS

A CLASS IS BASICALLY A STRUCT ON STEROIDS

AN OBJECT IS A VARIABLE OF THAT CLASS

"OBJECT" AND "CLASS" ARE POSSIBLY THE 2 MOST IMPORTANT WORDS IN PROGRAMMING

C++ RECAP

THIS GROUP OF VARIABLES & FUNCTIONS IS EFFECTIVELY A NEW, USER-DEFINED, TYPE

THIS USER DEFINED TYPE IS CALLED THE CLASS, AND IT CORRESPONDS EXACTLY TO A STRUCT IN C

C++ GOES FAR BEYOND C IN MAKING USER-DEFINED CLASSES FIRST CLASS TYPES, ON PAR WITH THE SYSTEM TYPES SUCH AS INT, FLOAT ETC

ANYONE CAN CREATE A VARIABLE OF THIS TYPE

A VARIABLE OF THIS CLASS IS CALLED AN OBJECT OF (OR AN INSTANCE OF) THE CLASS

DYNAMIC MEMORY ALLOCATION IN C++ IS VERY DIFFERENT THAN IN C

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

NEVER USE `malloc/free` AGAIN. EVER.

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE malloc OR free AGAIN, EVER.

RULE #2: USE new/delete FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE new[]/delete[] FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX new/delete AND new[]/delete[]

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

LET'S TAKE THEM ONE AT A TIME.

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

RULE #1: NEVER USE `malloc`/`free` AGAIN. EVER.

WHY? BECAUSE `malloc`/`free` ONLY ASSIGN MEMORY

RULE #1: NEVER USE `malloc`/`free` AGAIN. EVER.

WHY? BECAUSE `malloc`/`free` ONLY ASSIGN MEMORY

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

RULE #1: NEVER USE `malloc`/`free` AGAIN. EVER.

WHY? BECAUSE `malloc/free` ONLY ASSIGN MEMORY

THEY DON'T CALL CONSTRUCTORS AND DESTRUCTORS

USING THEM IN C++ WILL HAVE TERRIBLE TERRIBLE CONSEQUENCES

RULE #1: NEVER USE `malloc`/`free` AGAIN. EVER.

USING THEM IN C++ WILL HAVE TERRIBLE TERRIBLE CONSEQUENCES

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

ANYTIME YOU NEED TO CREATE A SINGLE VARIABLE OF A POINTER TYPE, JUST `new` TO BOTH ALLOCATE AND CONSTRUCT THE VARIABLE

YOU CAN USE THIS WITH SIMPLE TYPES (INT/FLOAT ETC) AS WELL AS WITH OBJECTS

YOU CAN ALSO PASS IN ARGUMENTS TO THE CONSTRUCTOR

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

ANYTIME YOU NEED TO CREATE A SINGLE VARIABLE OF A POINTER TYPE, JUST `new` TO BOTH ALLOCATE AND CONSTRUCT THE VARIABLE

YOU CAN USE THIS WITH SIMPLE TYPES (INT/FLOAT ETC) AS WELL AS WITH OBJECTS

YOU CAN ALSO PASS IN ARGUMENTS TO THE CONSTRUCTOR

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

`new` WILL FIRST ALLOCATE MEMORY AND THEN CALL THE CONSTRUCTOR FOR YOUR VARIABLE

`delete` WILL FIRST CALL THE DESTRUCTOR, AND THEN DEALLOCATE MEMORY

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

`delete` WILL FIRST CALL THE DESTRUCTOR, AND THEN DEALLOCATE MEMORY

IT GOES WITHOUT SAYING - NEVER USE THAT VARIABLE AFTER THE CALL TO `delete`

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

RULE #3: USE `new[]`/`delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

ANYTIME YOU NEED TO CREATE AN ARRAY OF ANY TYPE, JUST `new[]` TO BOTH ALLOCATE AND CONSTRUCT THE ARRAY

`new[]` WILL CYCLE THROUGH AND CALL THE NO-ARGUMENT CONSTRUCTOR FOR EACH ELEMENT OF THE ARRAY

YOU CAN'T PASS IN ARGUMENTS TO THE CONSTRUCTOR WITH `new[]`

RULE #3: USE `new[]`/`delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

ANYTIME YOU NEED TO CREATE AN ARRAY OF ANY TYPE, JUST `new[]` TO BOTH ALLOCATE AND CONSTRUCT THE ARRAY

`new[]` WILL CYCLE THROUGH AND CALL THE NO-ARGUMENT CONSTRUCTOR FOR EACH ELEMENT OF THE ARRAY

YOU CAN'T PASS IN ARGUMENTS TO THE CONSTRUCTOR WITH `new[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

`new[]` WILL FIRST ALLOCATE MEMORY FOR THE ARRAY, AND THEN CALL THE DEFAULT CONSTRUCTOR FOR EACH ARRAY ELEMENT. IT WILL TRACK ARRAY LENGTH TOO

`delete[]` WILL FIRST CALL THE DESTRUCTOR FOR EACH ARRAY ELEMENT, AND THEN DEALLOCATE MEMORY

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

`delete[]` WILL FIRST CALL THE DESTRUCTOR FOR EACH ARRAY ELEMENT, AND THEN DEALLOCATE MEMORY

IT GOES WITHOUT SAYING - NEVER USE THAT VARIABLE AFTER THE CALL TO

`delete[]`

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

`new` WILL FIRST ALLOCATE MEMORY AND THEN CALL THE CONSTRUCTOR FOR YOUR VARIABLE

`delete` WILL FIRST CALL THE DESTRUCTOR, AND THEN DEALLOCATE MEMORY

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

`new` WILL FIRST ALLOCATE MEMORY AND THEN CALL THE CONSTRUCTOR FOR YOUR VARIABLE

`delete` WILL FIRST CALL THE DESTRUCTOR, AND THEN DEALLOCATE MEMORY

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

`new` WILL FIRST ALLOCATE MEMORY AND THEN CALL THE CONSTRUCTOR FOR YOUR VARIABLE

`delete` WILL FIRST CALL THE DESTRUCTOR, AND THEN DEALLOCATE MEMORY

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

`new` WILL FIRST ALLOCATE MEMORY AND THEN CALL THE CONSTRUCTOR FOR YOUR VARIABLE

`delete` WILL FIRST CALL THE DESTRUCTOR, AND THEN DEALLOCATE MEMORY

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

`new` WILL FIRST ALLOCATE MEMORY AND THEN CALL THE CONSTRUCTOR FOR YOUR VARIABLE

`delete` WILL FIRST CALL THE DESTRUCTOR, AND THEN DEALLOCATE MEMORY

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

`new[]` WILL FIRST ALLOCATE MEMORY FOR THE ARRAY, AND THEN CALL THE DEFAULT CONSTRUCTOR FOR EACH ARRAY ELEMENT. IT WILL TRACK ARRAY LENGTH TOO

`delete[]` WILL FIRST CALL THE DESTRUCTOR FOR EACH ARRAY ELEMENT, AND THEN DEALLOCATE MEMORY

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

`new[]` WILL FIRST ALLOCATE MEMORY FOR THE ARRAY, AND THEN CALL THE DEFAULT CONSTRUCTOR FOR EACH ARRAY ELEMENT. IT WILL TRACK ARRAY LENGTH TOO

`delete[]` WILL FIRST CALL THE DESTRUCTOR FOR EACH ARRAY ELEMENT, AND THEN DEALLOCATE MEMORY

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX new/delete AND new[]/delete[]

ANYTHING YOU ALLOCATE+CONSTRUCT USING new, CLEAN UP USING delete

ANYTHING YOU ALLOCATE+CONSTRUCT USING new[], CLEAN UP USING delete[]

new[] WILL FIRST ALLOCATE MEMORY FOR THE ARRAY, AND THEN CALL THE DEFAULT CONSTRUCTOR FOR EACH ARRAY ELEMENT. IT WILL TRACK ARRAY LENGTH TOO

delete[] WILL FIRST CALL THE DESTRUCTOR FOR EACH ARRAY ELEMENT, AND THEN DEALLOCATE MEMORY

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

`new[]` WILL FIRST ALLOCATE MEMORY FOR THE ARRAY, AND THEN CALL THE DEFAULT CONSTRUCTOR FOR EACH ARRAY ELEMENT. IT WILL TRACK ARRAY LENGTH TOO

`delete[]` WILL FIRST CALL THE DESTRUCTOR FOR EACH ARRAY ELEMENT, AND THEN DEALLOCATE MEMORY

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX new/delete AND new[]/delete[]

ANYTHING YOU ALLOCATE+CONSTRUCT USING new, CLEAN UP USING delete

ANYTHING YOU ALLOCATE+CONSTRUCT USING new[], CLEAN UP USING delete[]

new[] WILL FIRST ALLOCATE MEMORY FOR THE ARRAY, AND THEN CALL THE DEFAULT CONSTRUCTOR FOR EACH ARRAY ELEMENT. IT WILL TRACK ARRAY LENGTH TOO

delete[] WILL FIRST CALL THE DESTRUCTOR FOR EACH ARRAY ELEMENT, AND THEN DEALLOCATE MEMORY

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

`new[]` WILL FIRST ALLOCATE MEMORY FOR THE ARRAY, AND THEN CALL THE DEFAULT CONSTRUCTOR FOR EACH ARRAY ELEMENT. IT WILL TRACK ARRAY LENGTH TOO

`delete[]` WILL FIRST CALL THE DESTRUCTOR FOR EACH ARRAY ELEMENT, AND THEN DEALLOCATE MEMORY

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new`/`delete` AND `new[]`/`delete[]`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new`, CLEAN UP USING `delete`

ANYTHING YOU ALLOCATE+CONSTRUCT USING `new[]`, CLEAN UP USING `delete[]`

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR

DESTRUCTORS ARE CRUCIAL WHEN YOUR CLASS HAS POINTERS OR FILE HANDLES AMONG ITS MEMBER VARIABLES.

In such cases, not freeing memory or closing files can lead to serious bugs - and memory and resource leaks.

DYNAMIC MEMORY ALLOCATION IN C++

IS VERY VERY SIMPLE - IF YOU REMEMBER 5 RULES

RULE #1: NEVER USE `malloc/free` AGAIN. EVER.

RULE #2: USE `new/delete` FOR SINGLE VARIABLES OF ALL TYPES.

RULE #3: USE `new[]/delete[]` FOR ARRAY VARIABLES OF ALL TYPES.

RULE #4: COROLLARY OF RULES 2 AND 3: NEVER MIX `new/delete` AND `new[]/delete[]`

RULE #5: CLEAN UP ALL POINTER MEMBER VARIABLES IN YOUR DESTRUCTOR