EE538: Computing Principles for Electrical Engineers

Lecture 2: A Tour of the C++ Language

University of Southern California

Instructor: Arash Saifhashemi

Hello World

Insertion stream operator

End of the stream

Standard C++ template library

```
#include <iostream>

// My first C++ program
/* Prints "Hello World"*/
int main() {
 std::cout << "Hello World" << std::endl;
 return 0;
}
```

Functions

Functions

- Can be overloaded
  - Same name
  - Different parameters

```
void PrintLine(int input) { std::cout << input << std::endl; }
void PrintLine(char input) { std::cout << input << std::endl; }
void PrintLine(float input) { std::cout << input << std::endl; }
void PrintLine(double input) { std::cout << input << std::endl; }
```

Operators

Operator	Type
++, --	Unary operator. **E.g.** i++
+, -, *, /, %	Arithmetic operators. **E.g.** a = b + c
<, <=, >, >=, ==, !=	Relational operators. **E.g.** a == b
&&, ||, !	Logical operators **E.g.** if (a == b && c == d)
&, |, <<, >>, ~, ^	Bitwise operators **E.g.** a << 2 , C = a & b
=, +=, *=, /=, %=	Assignment operators. **E.g.** a += 2;
?:	Ternary conditional operator. E.g. c == d ? 1 : 2;

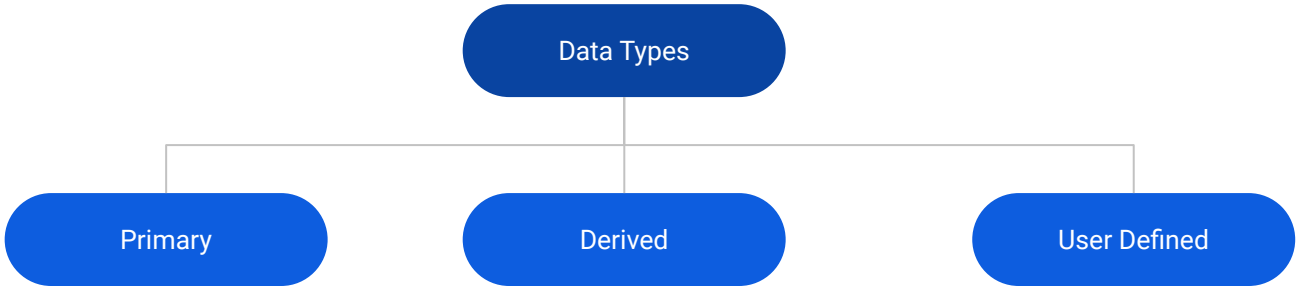
Operators

```
 a = 0101
 b = 1111

a&b     = 0110
a|b     = 1111
~a     = 1010
a^b     = 1010
a << 1 = 1010
a >> 2 = 0001
```

Data Types

-



- `int`
- `char`
- `bool`
- `float`
- `double`
- `void`

- `array`
- `pointer`
- `references`

- `struct`
- `class`
- `union`
- `enum`
- `typedef`

Variables

```
int main() {

 std::string hello = "Hello";
 std::string world = " world";
 int year = 2020;

 std::string hello_world = hello + world +
     " " + std::to_string(year);

 PrintLine(hello_world);
 return 0;
}
```

Variables and Modifiers

- `signed`
- `unsigned`
- `long`
- `short`

Note the that width is **machine dependant**.

The standard only guarantees the **minimum width**.

Type specifier	Equivalent type	Width in C++ standard
`short`	`short int`	at least **16**
`short int`		
`signed short`		
`signed short int`		
`unsigned short`	`unsigned short int`	
`unsigned short int`		
`int`	`int`	at least **16**
`signed`		
`signed int`		
`unsigned`	`unsigned int`	
`unsigned int`		
`long`	`long int`	at least **32**
`long int`		
`signed long`		
`signed long int`		
`unsigned long`	`unsigned long int`	
`unsigned long int`		
`long long`	`long long int` (C++11)	at least **64**
`long long int`		
`signed long long`		
`signed long long int`		
`unsigned long long`	`unsigned long long int` (C++11)	
`unsigned long long int`		

Machine Independent

- Fixed width integer types
  - Defined in header <cstdint>
    - `int8_t`
    - `int16_t`
    - `int32_t`
    - `int64_t`
    - `uint8_t`
    - `uint16_t`
    - `uint32_t`
    - `Uint64_t`

- Some defined constants
  - Defined in header <cstdint>
    - **`INT8_MIN`** and **`INT8_MAX`**
    - **`INT16_MIN`** and **`INT16_MAX`**
    - **`INT32_MIN`** and `INT32_MAX`
    - `INT64_MIN` and `INT64_MAX`
    - `UINT8_MAX`
    - `UINT16_MAX`
    - `UINT32_MAX`
    - `UINT64_MAX`

Float and Double

- Two primary types
  - **float** (4 byte)
  - **double** (8 byte)
  - **long double** (larger)
- No signed and unsigned variants.
  - They can represent negative numbers by default.

- Size
  - **float**: is a 32 bit IEEE 754 single precision Floating Point Number
    - 1 bit for the sign, (8 bits for the exponent, and 23 for the value)
    - float has 7 decimal digits of precision.
  - **double**: is a 64 bit IEEE 754 double precision Floating Point Number:
    - (1 bit for the sign, 11 bits for the exponent, and 52* bits for the value)
    - double has 15 decimal digits of precision.

Enum Class

- Scoped enumeration
  - Strongly typed
  - Strongly scoped
- Use **enum class** inste

```
// Enum type in C:
enum ColorPallet1 { Red, Green, Blue };
enum ColorPallet2 { Yellow, Orange, Red };

// Enum Class in C++
// Declaration
enum class ColorPalletClass1 { Red, Green, Blue };
enum class ColorPalletClass2 { Yellow, Orange, Red };

// Assignment
ColorPalletClass1 col1 = ColorPalletClass1::Red;
ColorPalletClass2 col2 = ColorPalletClass2::Red;
```

auto

```
int main() {
 std::vector<int> my_vector = {1, 2, 3, 4, 5, 6, 7, 8};
 for (auto n : my_vector) {
   n++;
 }
  return ;
}
```

I'm too lazy to specify the type,
please figure it out yourself!

Assignment

- Effectively copying

```
int i = 10;
```

Address	Value
0x5000	0
0x5004	10
0x5008	22

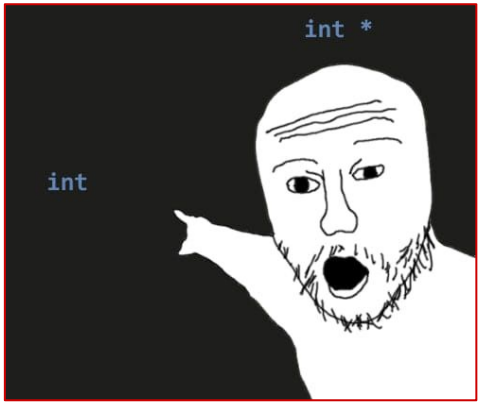
```
int j = i;
```

Address	Value
0x7000	55
0x8004	10
0x8008	45

What happens to j, when we do i=i+1 after the assignment?

Pointers

- Hold address of a variable
  - The value of p is 0x5004
  - p has its own address.




```
int i = 10;
```

```
int *p = &i;
```

Address	Value
0x5000	0
0x5004	10
0x5008	22

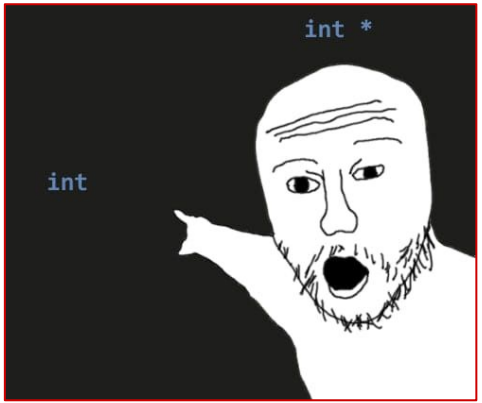
Can act as an **alias** to variables

What is the result of:

- `(*p)++; // i++`
- `p++;`

Pointers

- Hold address of a variable
  - The value of p is 0x5004
  - p has its own address.




```
int i = 10;
```

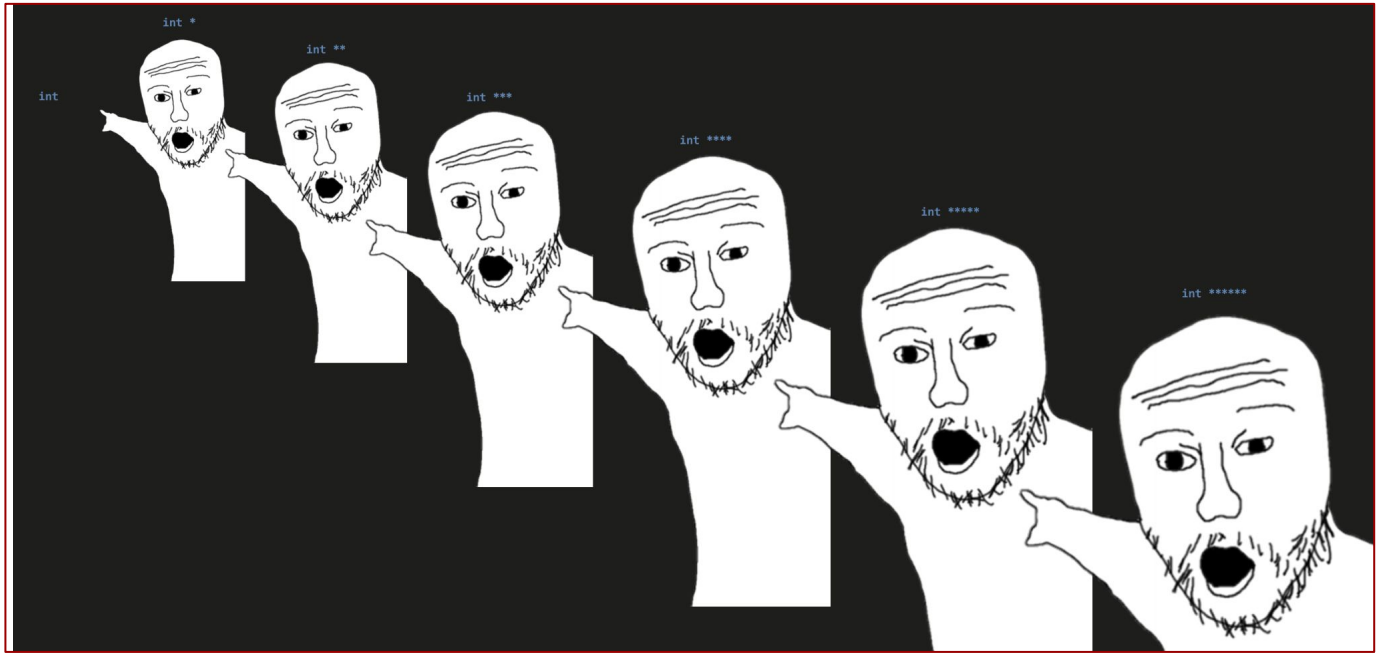
```
int *p = &i;
```

Address	Value
0x5000	0
0x5004	10
0x5008	22

Can act as an **alias** to variables

What is the result of:

- `(*p)++; // i++`
- `p++;`



- Pointer to Pointer
  - Used, but not very common.

```
int a = 1;
int *pp = &a;
int **r = &pp;
std::cout << "r: " << r << std::endl;
std::cout << "*r: " << *r << std::endl;
std::cout << "**r: " << **r << std::endl;
```

References

- An **alias** for a variable
  - Must be assigned **exactly once** upon definition `int i = 10;`

```
int &j = i;

int k = j;
k++
```

Address	Value
0x5000	0
0x5004	10
0x5008	10

What is the result of:

- `j++;`

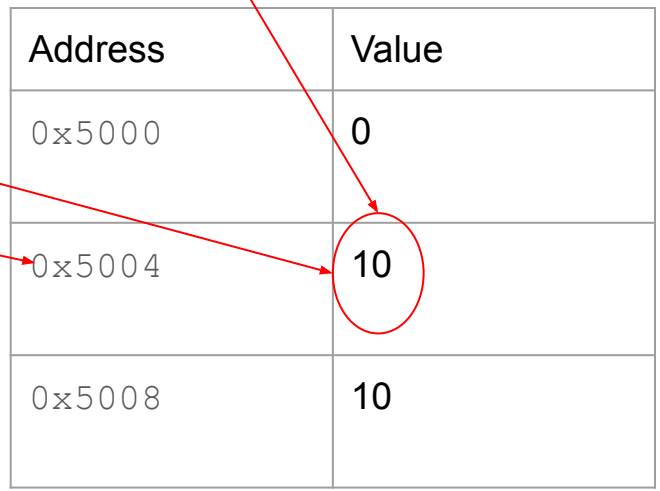
Pointer vs. Reference

- Sorry the syntax is confusing…

```
int i = 10;
```

```
int &j = i;

int *p = &i;
```



Address	Value
0x5000	0
0x5004	10
0x5008	10

An alias for a variable

- These do the same thing:
  - i++
  - j++
  - (*p)++

Passing Parameters

- Pass by value
  - Copies the original
- Pass by **reference**
  - Using a C++ reference
  - Using a pointer
    - Notice that we pass `&i`
  - When do we use it?
    - Modify the parameter
    - **Avoid** copying (It can be very slow)

```
// Acts like int j = i
void PassByValue(int j) {
  j++;
}
// Acts like int *j = &i;
void PassByReferenceUsingPointer(int *j) {
  (*j)++;
}
// Acts like int &j = i;
void PassByReferenceUsingReference(int &j) {
  j++;
}

int main() {
 int i = 10;
 PassByValue(i);
 PassByReferenceUsingPointer(&i);
 PassByReferenceUsingReference(i);
}
```

Why Using References (or Pointers)

- When is pass-by-value useful?
  - Avoiding modification
- Pass by Reference
  - Modification
    - In loops or functions
  - Return multiple variables
  - **Avoiding copying**

```
void PassByValue(std::vector<int> v)
```

```
// Acts like int j = i
void PassByValue(int j) {
    j++;
}
// Acts like int *j = &i;
void PassByReferenceUsingPointer(int *j) {
    (*j)++;
}
// Acts like int &j = i;
void PassByReferenceUsingReference(int &j){
    j++;
}
```

References in Loops

```
int main() {
 std::vector<int> my_vector = {1, 2, 3, 4, 5, 6, 7, 8};
 for (auto n : my_vector) {
   n++;
 }
 // What is the value of my_vector?
 for (auto &n : my_vector) {
   n *= 10;
 }
 return 0;
}
```

Pointers: new and delete

- Pointers can use **dynamic memory** AKA **heap**
  - **new**
  - **delete**
- **Heap** is a memory pool that you can dynamically access to.
- Depending on the inputs, the memory footprint changes
- Note: **new** and **delete** go hand in hand. If you **new** something, you should delete it.
  - **Memory leak**: a new without delete

```
int i = 5;
int *p = new int;
p = &i; // address of the new location is lost!
delete p;
```

```
int main() {

 int *p = nullptr;
 if (something) {
   p = new int;
   *p = 5;
   std::cout << "*p: " <<
     *p << std::endl;
 }

 if (p != nullptr) {
   delete p;
 }
}
```

Pointers

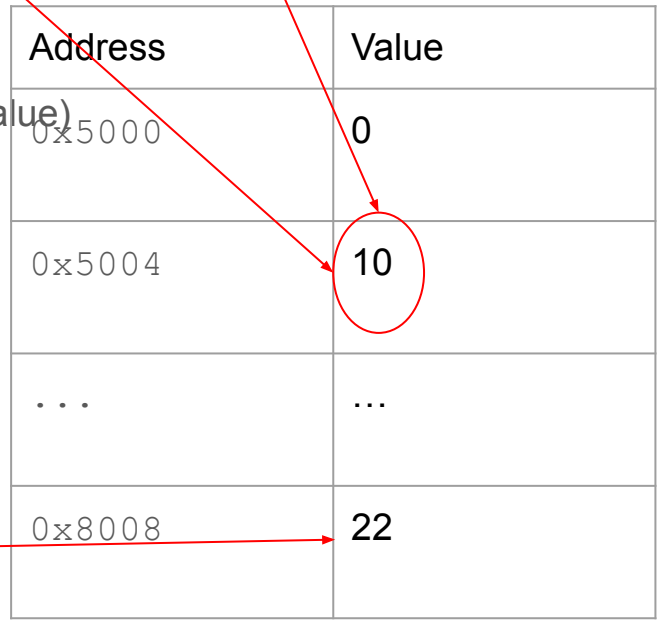
- We have a pointer either by:
  - Getting the address of a **currently defined variable**
  - Using the **new** operator
  - **Invalid** pointer: either
    - nullptr

```
int *p = &i; // p is pointing to a valid location
(*p)++; // Ok!
```

```
int *p = nullptr; // invalid memory location
(*p)++; // crash!
int *p = 0x12312; // Some random number that I made up and does not
belong to my program
(*p)++; // crash!
```

```
p = new int; // Valid memory location
*p = 22; // Ok!
(*p)++; // Ok!
```

```
int i = 10;
int &j = i;
```



Pointers

- Hold address of a variable
- We have a pointer either by:
  - Getting the address of a **currently defined variable**
  - Using the **new** operator
  - **Invalid** pointer: either

```
int *p = &i; // p is pointing to a valid location
(*p)++; // Ok!
```

```
int *p = nullptr; // invalid memory location
(*p)++; // crash!
int *p = 0x12312; // Some random number that I made up and does not
belong to my program
(*p)++; // crash!
```

```
int i = 10;
int &j = i;
```

value)

Address	Value
0x5000	0
0x5004	10
...	...

Pointers

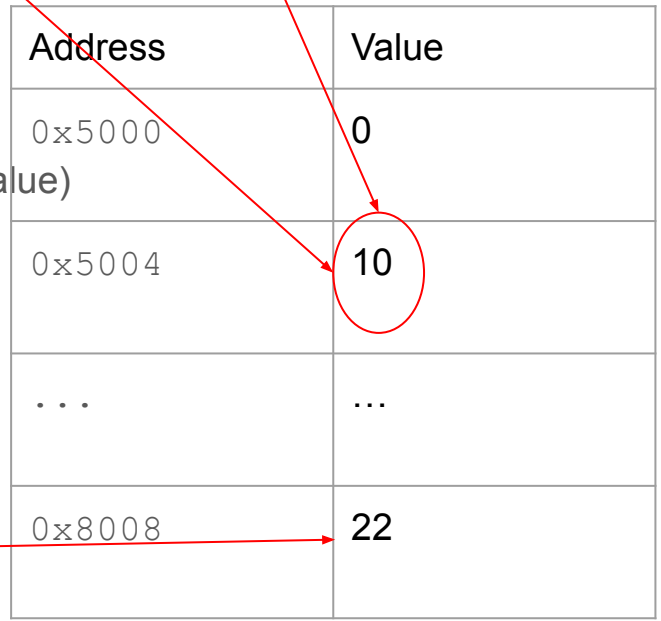
- Hold address of a variable
- We have a pointer either by:
  - Getting the address of a **currently defined variable**
  - Using the **new** operator
  - **Invalid** pointer: either

```
int *p = &i; // p is pointing to a valid location
(*p)++; // Ok!
```

```
int *p = nullptr; // invalid memory location
(*p)++; // crash!
int *p = 0x12312; // Some random number that I made up and does not
belong to my program
(*p)++; // crash!
```

```
p = new int; // Valid memory location
*p = 22; // Ok!
(*p)++; // Ok!
```

```
int i = 10;
int &j = i;
```



References v.s Pointers

- References:
  - Cannot be **reassigned**.
  - Must be **initialized** once defined.
  - Cannot be **NULL**.
  - References can become invalid
    - Less common but it can happen
- Pointers:
  - Need to be **dereferenced**.
    - `* or ->`
  - Limited **arithmetic** operations
  - Pointers use **new** and **delete** to store values in d
  - Are generally less **safe**

```
Person person;
 Person *person_ptr = &person;
 (*person_ptr).first_name = "Tommy";
 person_ptr->last_name = "Trojan";
```

- Pointers more likely to be misused, and they can be very dangerous.
- References can be misused too, but less likely

Be Careful with Pointers!

- **`Undefined Behavior!!`**
- When should we use them?
  - Try to **avoid** them if you can (If you can, use STL containers instead).
  - Use them when you need **dynamic memory allocation**.
  - For **pass-by-reference**, you should prefer **C++ references**.
- In what ways a pointer can be misused?
  - Creating undefined behavior when dereferencing a pointer pointing to an invalid location in the memory:
    - An **uninitialized** pointer
    - A **null** pointer
    - A pointer pointing to a location in the memory that **doesn't exist**.
    - A pointer that is **already deleted** (Don't delete a pointer twice!)
  - Memory leak:
    - Happens when you forget to delete a pointer.
    - E.g.: `int a = 5; int *p = new int ; p=&a;`

Remember: In C++ you should never access an **invalid memory** location.

Arrays

`int arr[8]`

- Features
  - Collection of items
  - Contiguous memory locations
  - Can be indexed
- What you need to know:
  - Index starts from 0!
  - Array size is predefined
    - Unless it is a dynamic array

```
int main() {
 int arr[8];
 arr[0] = 5;
 arr[2] = -10;
 print_array(arr, 8);

 int x = arr[0];
 std::cout << "x: " << x << std::endl;

 return 0;
}
```

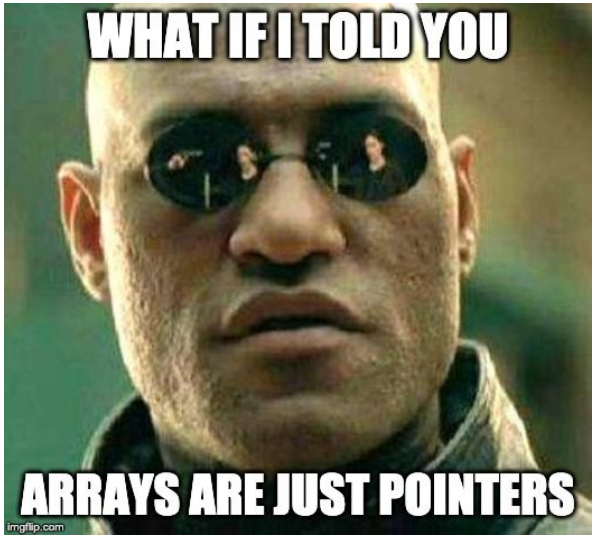
Arrays

`int arr[8]`


```
53    int my_arr[8];
54    InitializeArray(my_arr, 8);
55    PrintArray(my_arr, 8);
56    int *p = &my_arr[0];
57    std::cout << "arr: " << my_arr << std::endl;
58    std::cout << "p: " << p << std::endl;
59    std::cout << "*(p): " << *(p) << std::endl;
60    std::cout << "*(p+1): " << *(p + 1) << std::endl;
61    std::cout << "*(p+2): " << *(p + 2) << std::endl;
62
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

```
{ 0, 1, 2, 3, 4, 5, 6, 7 }
arr: 0x7ffee529a6d0
p: 0x7ffee529a6d0
*(p): 0
*(p+1): 1
*(p+2): 2
```



Arrays

`int arr[8]`

- Features
  - Collection of items
  - Contiguous memory locations
  - Can be indexed
- What you need to know:
  - Index starts from 0
  - Array size is predefined
    - Unless it is a dynamic array
  - It is really a pointer
    - It can be dangerous!
    - It is passed by reference

```
int main() {
 int arr[8];
 arr[0] = 5;
 arr[2] = -10;
 print_array(arr, 8);



 int x = arr[0];
 std::cout << "x: " << x << std::endl;


 return 0;
}
```

Dynamic Arrays

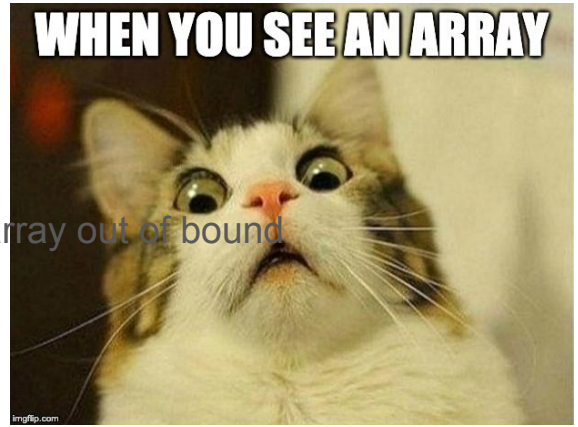
- Dynamic array is just a pointer that points to the start of **a block of memory**.
- We use `new Type[]` and `delete[]`
  - The **size** of dynamic array **doesn't have to be known** at compile time
  - But it still **cannot be resized**!

```
arr = new int[size];

delete [] arr;
```

Array Misuse

- Index out of bound
  - Undefined behavior
  - Always check to see if there is any chance you index array out of bound



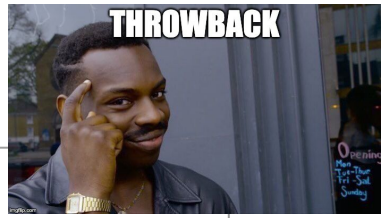

```
int arr[10];
for (int i = 0; i <= 10; i++) {
  arr[i] = 0;
}
```

Remember: In C++ you should never access an **invalid memory** location.

std::vectors

- An enhanced version of Arrays
- Important methods to know
  - push_back()
  - pop_back()
  - insert()
  - erase()
- Homework:
  - What is the time complexity of each of the above functions?

Why Vectors?



Data type	Feature
Array	• Size is **fixed** at compile time • Cannot be **resized** • There is not a reliable way to **find the array size** ○ Homework: how do we find the size of an array? • Can be misused (out of bound)
Dynamic Array	• Size **doesn't have to be known** at compile time • Cannot be **resized** • There is no way to find the size. • Can be misused (out of bound)
Vectors	• Size **doesn't have to be known** at compile time • Can be **resized** automatically • The size is always **kept updated** • Can be misused (out of bound) with [ ] ○ Misuse can be controlled using **at()** method (homework)

Flow control

- Conditionals
  - if
  - switch
- Loops
  - while
  - do-while
  - For

- See src/control in cpp_tour repo for examples

Flow control

```
int i = 0
while (i < 10) {
   if (i == 2) {
     continue;
   }
   my_vector.push_back(i);
   i++;
}
```

```
bool b = false;
 if (b = true) {
   std::cout << "b is true" << std::endl;
 }
```

- Beware of corner case conditions
- Beware of infinite loops

std::string

- Important methods to know
  - push_back()
  - pop_back()
  - getline()
  - concat
  - insert()
- Homework:
  - What is the time complexity of each of the above functions?