CS4414 Recitation 2

C++ Types and Containers

02/03/2023

Alicia Yang

C++ Built-in Types

C++ is strongly typed

- A **declaration** is a statement that introduce a name to the program with a specified type

```
int  x ;            // declaration
```

type

variable

C++ is strongly typed

- A **declaration** is a statement that introduce a name to the program, with a specified type

```
int  x ;                  // declaration
```

- A **declaration** can also follow with an **initialization**

```
int  x = 5;               // declaration + initialization
```

type

variable

Initial value

C++ is strongly typed

- A **declaration** is a statement that introduce a name to the program with a specified type

```
int  x ;                    //  declaration
```

- A **declaration** can also follow with an **initialization**

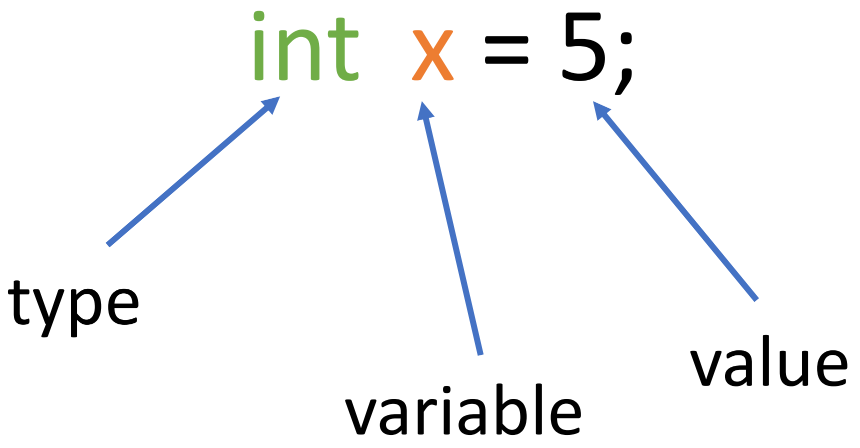
```
int  x = 5;                 // declaration + initialization
```

- Later, you can use variable x in expressions such as

```
int  y = x + 1;             // initialization of y using x
x = 7;                      // reassignment
```

C++ is strongly typed

- A C++ variable has a name, a type, a value and an address in memory
  - A type: defines a set of **possible values** and **operations** that this variable can do
  - A value: a set of bits to be interpreted by its type
  - An object: some memory that holds a value of some type



C++ types

- Primitive(fundamental) data types
  - bool
  - char
  - int
  - float
  - double

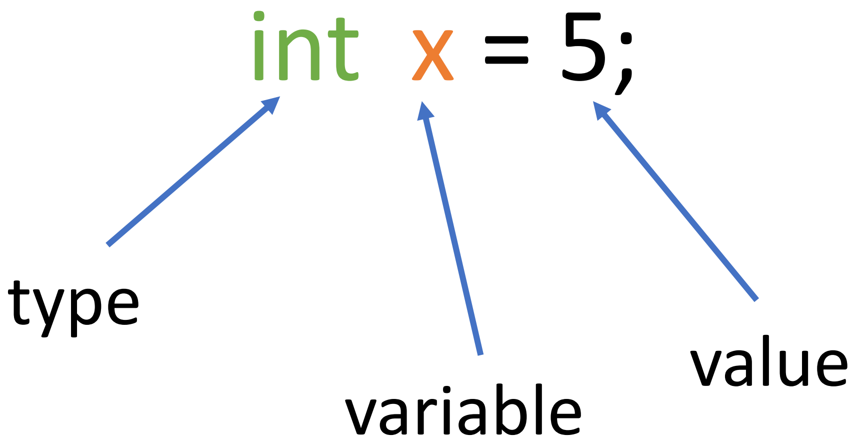
- Derived data types
  - pointer
  - array
  - function
- User-defined data types
  - class
  - struct

C++ types

- bool // boolean, possible values are true and false
- char // character, possible values are 'a', 'z', '9', '\'' ..
- int // integer, possible values are 36, -273, 10006, ..
- double // double-prevision floating-point number, possible values are 3.14, 230421.0, ..
- unsigned // non-negative integer, possible values are 0, 365,…
- uint8_t // 8-bit(1-byte) unsigned integer, possible values are 0, .. 200, .. 255

C++ is strongly typed

- A C++ variable has a name, a type, a value and an address in memory
  - A type: defines a set of **possible values** and **operations** that this variable can do
  - A value: a set of bits to be interpreted by its type
  - An object: some memory that holds a value of some type

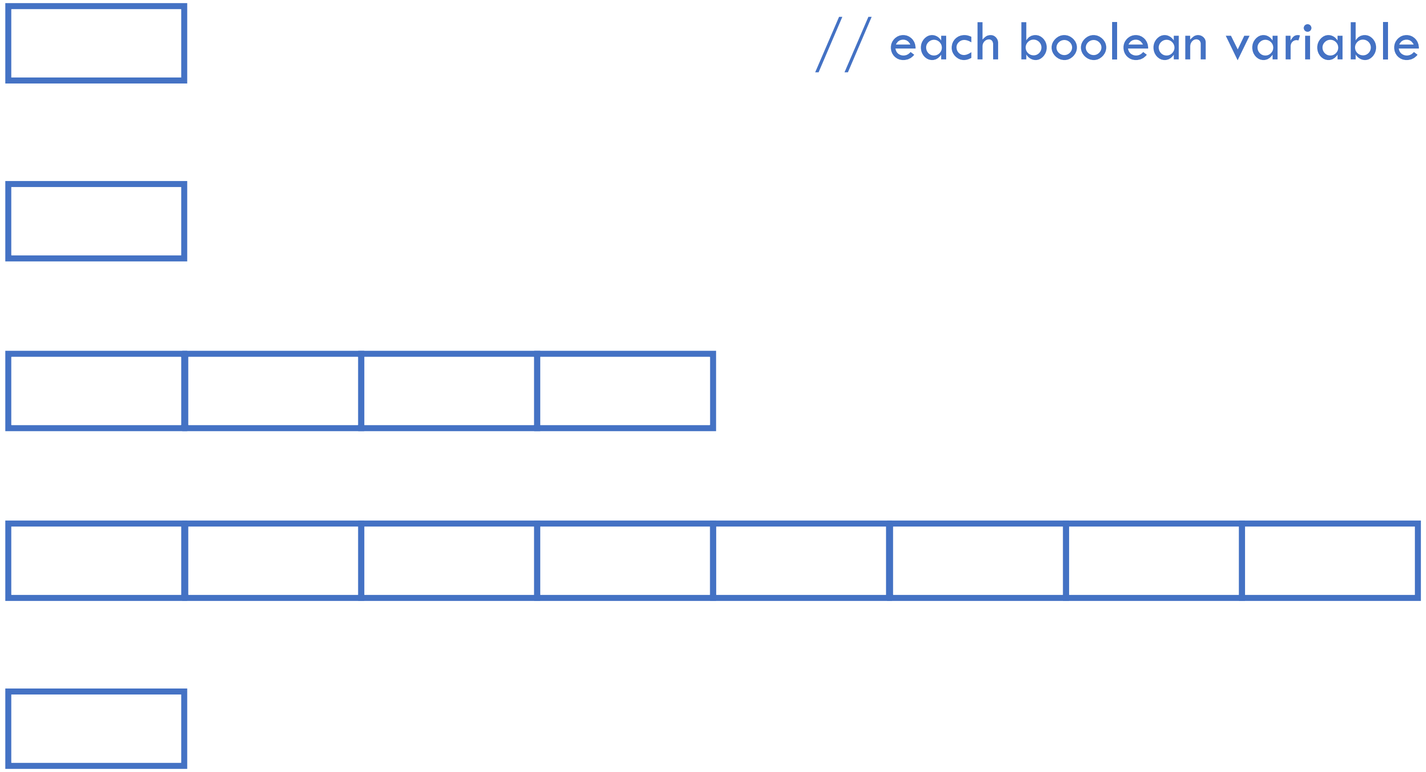


C++ fundamental data type

- Lots of integer types
  - int, short, unsigned int, long, long long, unsigned long, …
  - Even more: int8_t, int16_t, int32_t, int64_t, …

C++ fundamental type correspond to fixed sizes

- bool // each boolean variable has 1 byte(8 bit)
- char
- int
- double
- uint8_t



C++ fundamental data type

- How do I find out the **size of a built-in type?**
  - Use the built-in function sizeof(variable name) or sizeof(<type>) to find out the size of the variable's **type**

```
long long int x = 0;
std::cout << sizeof(x) << std::endl;                  // print 8
std::cout << sizeof(long long int) << std::endl;      // print8
```

Question: What is the largest value that a 4-byte integer can represent?

Question: What is the largest value that a 4-byte integer can represent?

- 4 bytes = 32 bits

  A 32-bit datatype can represent $\mathbf{2^{32}}$ distinct values
- A signed 4-byte integer can represent numbers from $-\mathbf{2^{31}}$ (-2,147,483,648) to $\mathbf{2^{31}-1}$ (2,147,483,647)
- An unsigned 4-byte integer can represent numbers from 0 to $\mathbf{2^{32}-1}$ (4,294,967,295)
- **Tip:** Use fixed-width integer types defined in **cstdint.** 4-byte integers for normal use(int32_t, uint32_t) and 8-byte integers(int64_t, uint64_t) for representing larger values

Operators defined by types

- Arithmetic: a + b, a – b, a * b, …
- Logical: !a, a&&b, a || b
- Relational: a == b, a < b, a > b, a <=b, …
- Assignment: a = b, a += b, a /= b, …
- Increment: ++ a, --a, a++, a--

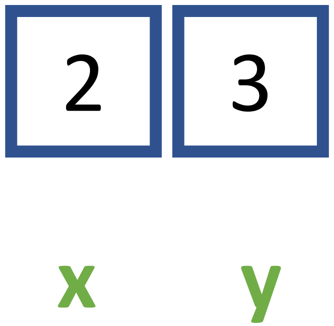
```
if (x + y < 7 && !(z > 10)){
        // do something
}
```

+=, -=, *=, /=, …

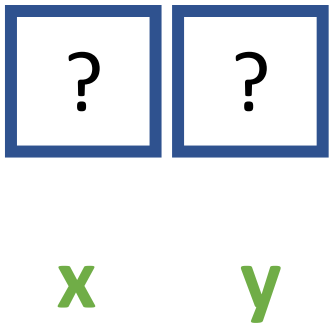
- x += y is equivalent to writing x = x + y
- Can also use for bools: b1 |= b2

More on increment and decrement

- Pre-increment (**++a**) and post-increment (**a++**) behave differently

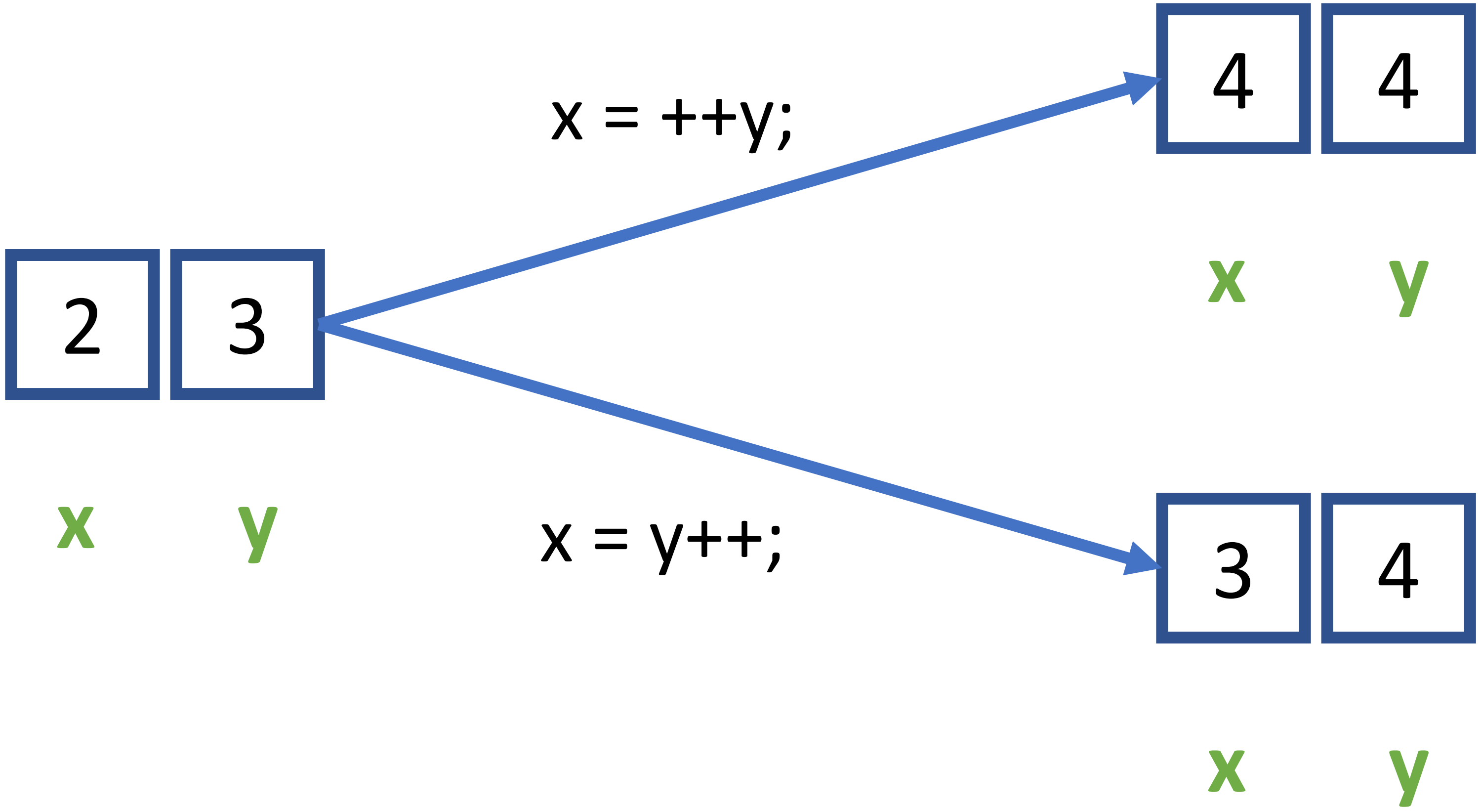


x = ++y;
or
x = y++;



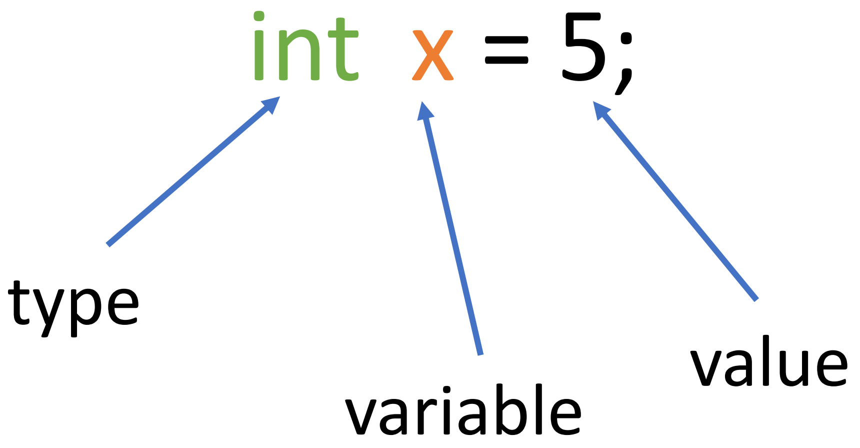
More on increment and decrement

- Pre-increment (**++a**) and post-increment (**a++**) behave differently



C++ is strongly typed

- A C++ variable has a name, a type, a value and an address in memory
  - A value: a set of bits to be interpreted by its type
  - A type: defines a set of possible values and operations that this variable can do
  - **An object: some memory that holds a value of some type**



Address and initial value

- Can obtain the **address** (represented in hex) with the **&** operator

```
std::cout <<  &x << std::endl;
                                        // prints 0x7ffd55bdaa4
```

Address and initial value

- Can obtain the **address** (represented in hex) with the **&** operator

```
std::cout <<  &x << std::endl;
                                        // prints 0x7ffd55bdaa4
```

- What happens if you use an **uninitialized** variable**?**

```
int  x ;
std::cout << x << std::endl;
```

Address and initial value

- Can obtain the **address** (represented in hex) with the **&** operator

```
std::cout <<  &x << std::endl;
                          // prints 0x7ffd55bdaa4
```

- What happens if you use an **uninitialized** variable?

```
int  x ;                       // uninitialized value
std::cout << x << std::endl;
                          // the value of x is undefined
```

Implicit conversion

- False is 0, true is 1. Any non-zero int is true, int 0 is false.

```
if (my_int) {}        // equivalent to if (my_int != 0)
```

- Implicit conversion from char to int (use ASCII code)

```
isdigit(ch): ch >= 48 && ch <= 57
```

Implicit conversion

- False is 0, true is 1. Any non-zero int is true, int 0 is false.

```
if (my_int) {}        // equivalent to if (my_int != 0)
```

- Implicit conversion from char to int (use ASCII code)

```
isdigit(ch): ch >= 48 && ch <= 57
```

- Written better as,

```
isdigit(ch): ch >= ‘0’ && ch <= ‘9’
```

demo

C++ auto keyword and const qualifier

- Compiler infers type of variable defined with the auto keyword

```
int max(int x, int y);      // function declaration
auto m = max(x, y);         // m is an int,
                            // the return type of m of max()
```

- const keyword before a variable declaration fixes its value to the initial value

```
const double pi = 3.14;  // good for readability
```

Exercise: Explain the error

```cpp
#include <iostream>

class myClass {
public:
  void print () {
    std::cout << "My integer is: " << myInt << std::endl;
  }

private:
  int myInt = 10;
};


int main() {
  const myClass myObj;
  myObj.print();
}
```

```
~ $
~ $ g++ program.cpp -o program
program.cpp: In function ‘int main()’:
program.cpp:16:15: error: passing ‘const myClass’ as ‘this’ argument discards qualifiers [-fpermissive]
   16 |   myObj.print();
      |               ^
program.cpp:5:8: note:   in call to ‘void myClass::print()’
    5 |   void print () {
      |        ^~~~~
~ $
```

Exercise: Explain the error

```cpp
#include <iostream>

class myClass {
public:
  void print () {
    std::cout << "My integer is: " << myInt << std::endl;
  }

private:
  int myInt = 10;
};


int main() {
  const myClass myObj;
  myObj.print();
}
```

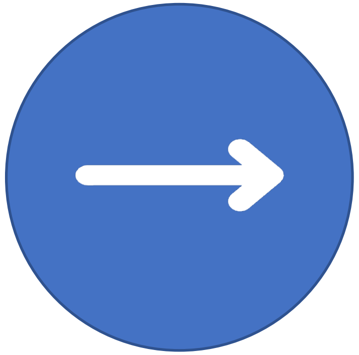
```
~ $
~ $ g++ program.cpp -o program
program.cpp: In function 'int main()':
program.cpp:16:15: error: passing 'const myClass' as 'this' argument discards
qualifiers [-fpermissive]
   16 |   myObj.print();
      |               ^
program.cpp:5:8: note:   in call to 'void myClass::print()'
    5 |   void print () {
      |        ^~~~~
~ $
```

- Print function can potentially change the state of a myClass Object, so it cannot be called on a const object
- To assert that print cannot change object state, change it to void print () const {}

Follow up: What happens when myInt is incremented in the const print function?

```
~ $
~ $ g++ program.cpp -o program
program.cpp: In member function ‘void myClass::print() const’:
program.cpp:7:5: error: increment of member ‘myClass::myInt’ in read-only obje
ct
    7 |     myInt++;
      |     ^~~~~
~ $
```

More in future recitations



POINTERS

CLASSES

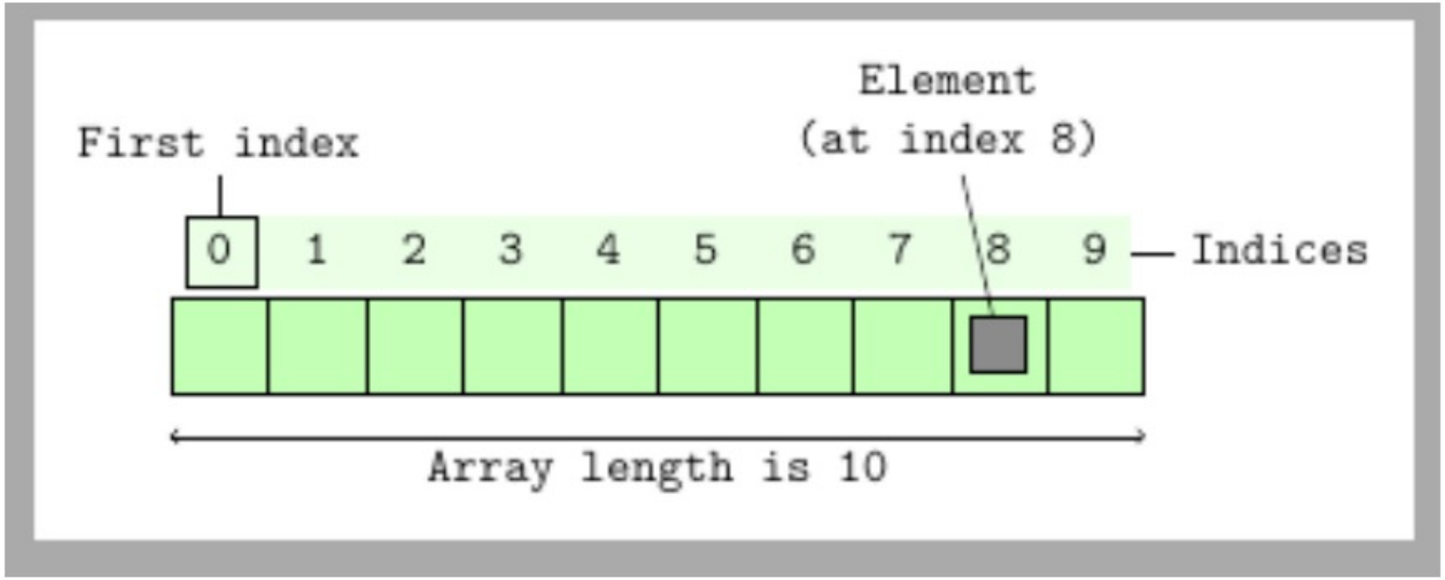
C++ Containers

C++ Container

- A Container is an object used to **store other objects** and take care of the **management of the memory** of the objects it contains.

- Containers include many commonly used structure:
  - std::array,
  - std::vector,
  - std::queues,
  - std::map,
  - std::set,
  - …

Array – a fundamental data type



- Arrays must be declared by type and size
- The size must be fixed at compile-time
- Stores elements contiguously (in continuous memory locations)
- Elements are accessed starting with position 0 (0-based indexing)
- O(1) access given the index of the element

C-style array (raw array)

- C-style array is a block of memory that can be interpreted as an array

```
int  a[10];
```

// declare **a** as an **array object** that consist of 10 **contiguous allocated** objects of **type int**

```
int  a[3] = {1 , 3,  6} ;
```

// assignment of objects in array

a	1	3	6

std::array<T, N> ---a container that holds fixed size arrays

- Has the same semantics as a C-style array, but implemented by standard template library
- To use this container, include it at the beginning of the file

```
#include <array>
```

- T and N are template parameters: T is the type of the array, and N defines the number of elements
  - E.g., std::array<char, 10>, std::array<int, 3>

std::array<T, N> ---a container that holds fixed size arrays

- Has the same semantics as a C-style array, but implemented by standard template library
- To use this container, include it at the beginning of the file

```
#include <array>
```

- T and N are template parameters: T is the type of the array, and N defines the number of elements
- **Why do we want to use std::array offered by C++ Standard Template Library(std)?**

C-style array vs. std::array<T, N>

demo

- C-style array Notes
  - No bound check when accessing element using operator[]
    - Undefined result if access a[20] if a is an array with size 3
  - Array-to-pointer decay
    - E.g., When pass a C-style array as **a value** to a function it decays to **a pointer** of the first element in the array, losing the size information.

C-style array vs. std::array<T, N>

demo

- C-style array characteristics
  - No bound check when accessing element using operator[]
  - Array-to-pointer decay

```
void print_array(int arr[]){
    size_t arr_size = sizeof(arr) / sizeof(int)
    for(int i = 0; i < arr_size; ++ i){
        std::cout << arr[i] << std::endl;
    }
}
```

```
void print_array(int * arr){
    size_t arr_size = sizeof(arr) / sizeof(int)
    for(int i = 0; i < arr_size; ++ i){
        std::cout << arr[i] << std::endl;
    }
}
```

```
yy354@en-ci-cisugcl14:~/CS4414Demo/recitation2$ g++ -fstack-protector-all array_example.cpp -o arr
array_example.cpp: In function 'void print_arr(int*)':
array_example.cpp:11:34: warning: 'sizeof' on array function parameter 'arr' will return size of 'int*' [-Wsizeof-array-argument]
   11 |       size_t arr_size = sizeof(arr) / sizeof(int);
      |                                  ^
array_example.cpp:10:20: note: declared here
   10 | void print_arr(int arr[]){
      |                ~~~~^~~~~
```


https://cppinsights.io

C-style array vs. std::array<T, N>

demo

Std::array<T> has more functions of standard container, makes it easier to use

```
std::array<int, 3> a = {1, 2, 3};
```

- size() : get the size of the array

```
std::cout << a.size() << std::endl;
```

- at() : access specified element with bounds checking

```
std::cout << a.at(2) << std::endl;
```

- Use iterator to access container elements

```
for(auto it = a.begin(); it < a.end(); ++it )
{....}
```

- More functionalities: https://en.cppreference.com/w/cpp/container/array

std::vector<T>

- T is a template parameter
- Std::vector<int> is a vector of integers, std::vector<char> is a vector of characters
- Same as std::array, T can be a class or other C++ container
  - E.g., std::vector<std::vector<int>> ,

    std::vector<std::map<int, std::string>>…

std::vector<T>

- T is a template parameter
- Std::vector<int> is a vector of integers, std::vector<char> is a vector of characters
- Same as std::array, T can be a class or other C++ container
  - E.g., std::vector<std::vector<int>> ,

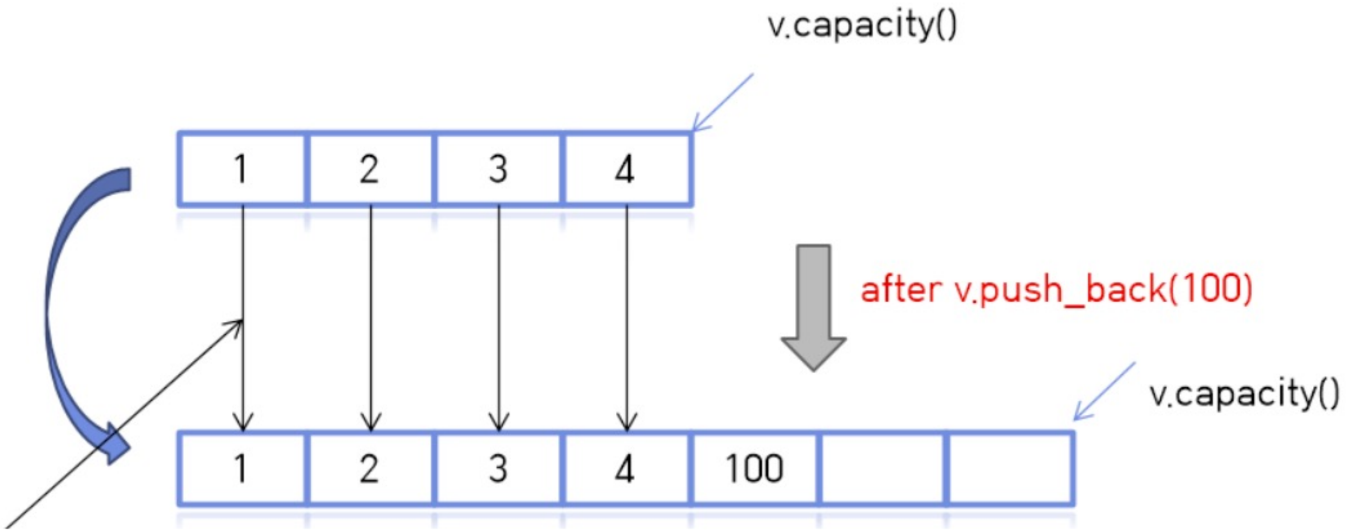
Why do we want to use std::vector<T> ?

std::vector<T> - A dynamic-sized array

- Main problem: How to support inserting elements efficiently?
- Concept of size vs. capacity

std::vector<T> - A dynamic-sized array

- Main problem: How to support inserting elements efficiently?
- Concept of size vs. capacity
- Reallocates elements when capacity is exceeded



Complexity of std::vector<T>::push_back

- Most push_backs will be O(1) (when size < capacity)
- Some will have linear complexity (when the vector is reallocated)
- Amortized O(1) complexity with exponential growth in capacity
- What about the complexity of inserting at a random position in the vector?

Complexity of std::vector<T>::push_back

- Most push_backs will be O(1) (when size < capacity)
- Some will have linear complexity (when the vector is reallocated)
- Amortized O(1) complexity with exponential growth in capacity
- What about the complexity of inserting at a random position in the vector?

```
std::vector<T>::insert(iterator pos, const T& value)
```

Must shift elements to the right! Linear complexity

Exercise

- Pick a large N (> 1 million)
- Program A: Creates a vector of N elements and assigns vec[i] = i for each i in a for-loop
- Program B: Creates an empty vector and calls vec.push_back(i) N times in a for-loop
- Program C: Creates an empty vector and calls vec.insert(vec.begin(), N-i-1) N times in a for-loop
- Measure the time taken by program A, B and C

Reference

- Effective C++: 55 specific ways to improve your programs and designs, Scott Meyers, 3rd edition
- A Tour of C++, Bjarne Stroustrup
- Large Scale C++, Process and Architecture, John Lakos, Volume 1
- C-style array cppreference: https://en.cppreference.com/w/cpp/language/array
- Container reference: https://cplusplus.com/reference/stl/
- std::array documentation: https://en.cppreference.com/w/cpp/container/array
- std::vector documentation: https://cplusplus.com/reference/vector/vector/
- CS4414 recitation slides, from Sagar Jha, TA for this course in 2020, 2021