Introduction to Programming

Chapter 9

User-Defined Types

Types in C++

Recall that a data type is a set of values and set of operations on those values

Fundamental types

- values immediately map to machine representations
- operations immediately map to machine instructions.

Compound types

- New types can be created in C++
- C++ standard library provides useful types: string, vector, …
- We can build our own types

Creating new types

Using `struct` we can define a new data-type as collection of variables of different types.

Example: A `student` type with three variables

- name of type `string`
- age of type `int`
- gpa of type `double`

```
struct student {
    string name;
    int age;
    double gpa;
};
```

Variables of type student can be declared

```
student s1, s2;
```

Each variable of type `student` has it’s own name, age, and gpa field

Fields can be accesses using dot operator

```
s1.name = "Ali";
s1.age = 20;
s1.gpa = 3.5;
```

```
s2.name = "Ahmed";
s2.age = 21;
s2.gpa = 3.2;
```

Initializing structures

= sign is optional since C++11

Using initializing list

```
student s1 = {"Ahmed", 20, 3.2};

student s2 {"Na-laiq", 19};
```

s3.gpa will be 0.0
(default value for double)

Designated initializers (C++20)

```
student s3 {.name = "Ali", .age = 21, .gpa = 3.8};

student s4 {.name = "Prodigy", .gpa = 3.7};

student s5 {.gpa = 2.2, .name = "Nobody"};
```

s3.age will
be 0

error, designator order must match declaration order

Passing `struct` to functions

Values of type student can be passed to functions just like any other values

Pass by value

```
void print(student s)
{
    cout << s.name << " " << s.age << " " << s.gpa << endl;
}
```

Pass by reference
(using C++ reference)

```
void print(const student& s)
{
    cout << s.name << " " << s.age << " " << s.gpa << endl;
}
```

Pass by reference
(using C++ pointers)

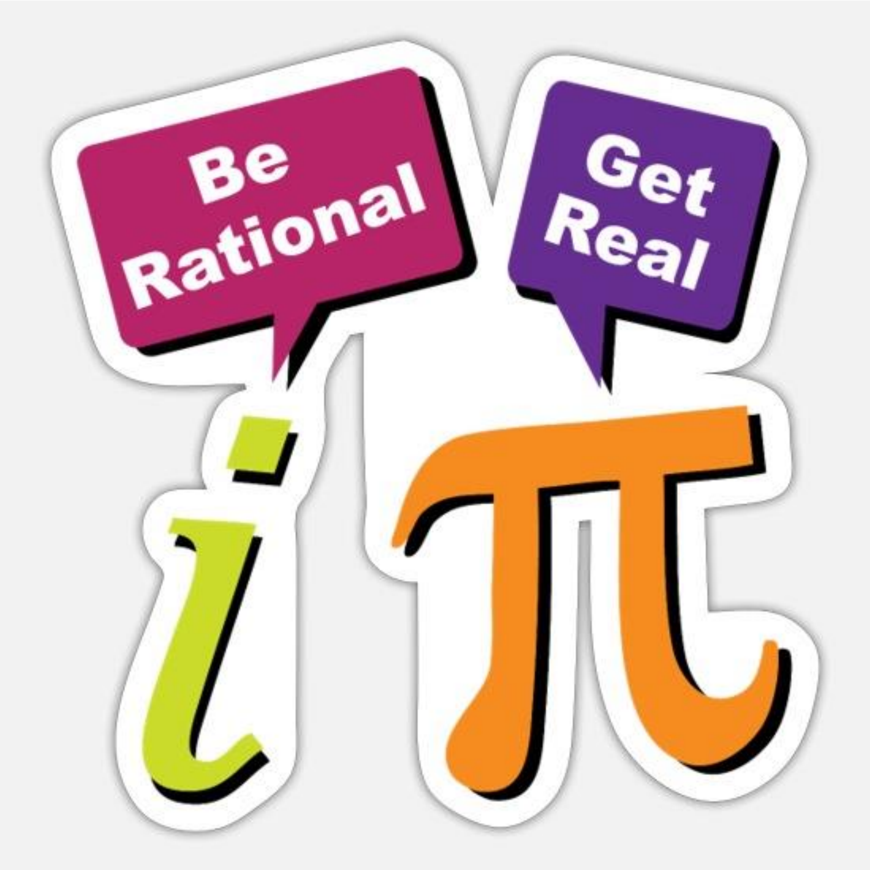
```
void print(student* s)
{
    cout << s->name << " " << s->age << " " << s->gpa << endl;
}
```

`->` oerator to dereference and access a field through a pointer
`s->name` is same as `(*s).name`

Returning `struct` from functions

A structure can be returned from functions just like any other value

```
student create_student(string name, int age, double gpa)
{
    student s;
    s.name = name;
    s.age = age;
    s.gpa = gpa;
    return s;
}
```



Example: A data type for Complex Numbers

Crash course in complex numbers

A complex number is a number of the form $a + bi$ where a and b are real and $i \stackrel{\text{def}}{=} \sqrt{-1}$.

Complex numbers are a *quintessential mathematical abstraction* that have been used for centuries to give insight into real-world problems not easily addressed otherwise.

To perform *algebraic operations* on complex numbers, use real algebra, replace i^2 by -1 and collect terms.

- Addition example: $(3 + 4i) + (-2 + 3i) = 1 + 7i$.
- Multiplication example: $(3 + 4i) \times (-2 + 3i) = -18 + i$.

The *magnitude* or *absolute* value of a complex number $a + bi$ is $|a + bi| = \sqrt{a^2 + b^2}$

Example:
$|3 + 4i| = 5$

Applications: Signal processing, control theory, quantum mechanics, analysis of algorithms...

A data type for complex numbers

A complex number is a number of the form $a + bi$ where a and b are real and $i \stackrel{\text{def}}{=} \sqrt{-1}$.

Representing complex values

```
struct complex {
    double re;
    double im;
};
```

Operations on complex values: functions that accept and return complex values

add operation

```
complex add(const complex& a, const complex& b) {…}
```

using add()

```
complex a{3,2}, b{-1,3};
complex c = add(a,b);
```

overload + operator for complex

```
complex operator+(const complex& a, const complex& b) {…}
```

```
complex c = a + b;
```

← equivalent to `c = operator+(a,b)`

A data type for complex numbers

values:

```
struct complex {
    double re;
    double im;
};
```

operations:

assuming a, b are variable of type complex

operation	function implementing the operation	use example
+	`complex operator+(const complex& a, const complex& b)`	`complex c = a + b`
+=	`complex operator+=(complex& a, const complex& b)`	`a += b`
*	`complex operator*(const complex& a, const complex& b)`	`complex c = a * b`
*=	`complex operator*=(complex& a, const complex& b)`	`a *= b`
==	`bool operator==(const complex& a, const complex& b)`	`if(a==b) {…}`
abs()	`double abs(const complex& a)`	`double x = abs(a)`

Implementing `complex` operations

complex addition

```
complex operator+(const complex& a, const complex& b) {
    complex c = complex {a.re + b.re, a.im + b.im};
    return c;
}
void operator+=(complex& a, const complex& b) {
    a.re += b.re;
    a.im += b.im;
}
```

complex multiplication

```
complex operator*(const complex& a, const complex& b) {
    double real = a.re * b.re - a.im * b.im;
    double imag = a.re * b.im + a.im * b.re;
    return {real, imag};
}
void operator*=(complex& a, const complex& b) {
    a = a * b;
}
```

Implementing `complex` operations

magnitude

```
double abs(const complex& a) {
    return sqrt(a.re*a.re + a.im*a.im);
}
```

output stream operator

```
ostream& operator<<(ostream& out, const complex& a) {
    out.precision(2);
    out << std::fixed << a.re << " + " << a.im;
    return out;
}
```

A client for `complex` data type

test client

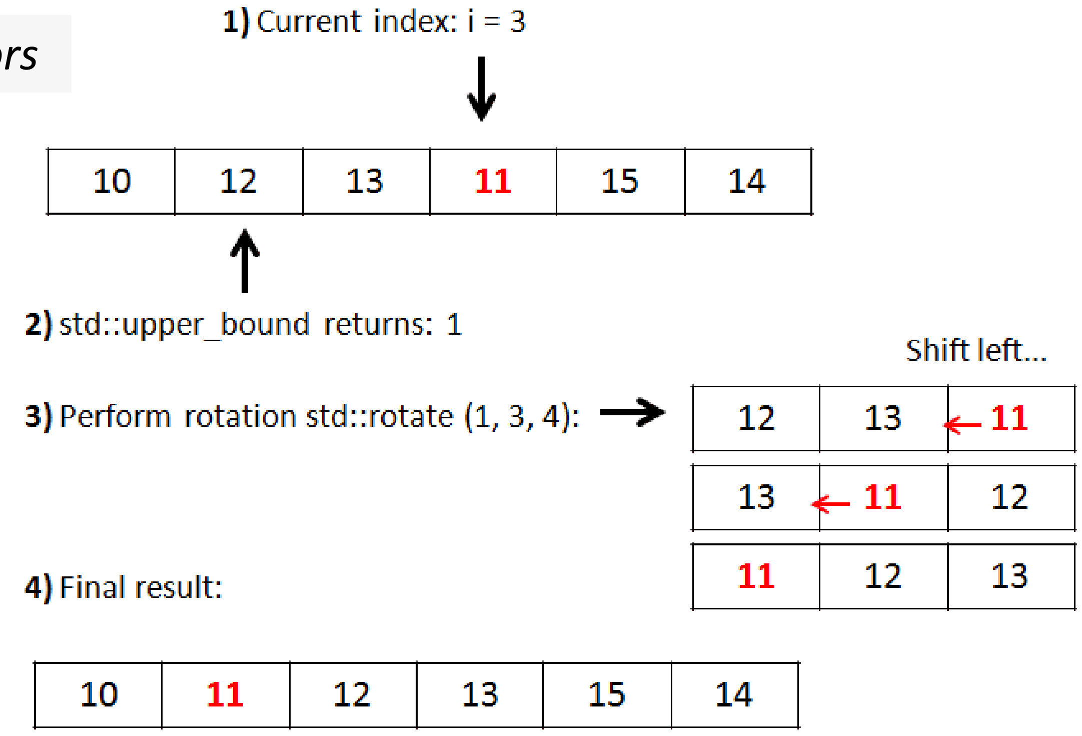
```
int main() {
    complex a{2.5,2}, b{-1,3};
    cout << "a: " << a << "\n";
    cout << "b: " << b << "\n";
    cout << "a+b: " << (a+b) << "\n";
    cout << "a*b: " << (a*b) << "\n";

    a *= b;
    cout << "a after exceuting a*=b: " << a << "\n";

    cout << "Magnitude of a: " << abs(a);
}
```

Standard Template Library

Standard Template Library

Part of the C++ Standard Library ← installed with the compiler, no need to install separately

Four components: *algorithms*, *containers*, *functions*, and *iterators*



Example: Insertion sort using STL

```
for (auto i = start; i != end; ++i)
    std::rotate(std::upper_bound(start, i, *i), i, std::next(i));
```

Containers

Container: A data type to store a collection of values (similar to arrays)

Sequence containers

array<T, N>	static array of values of type T of fixed size N (N known at compile time)
vector<T>	dynamic array of values of type T (resizable)
dequeue<T>	double ended queue
list<T>	linked list

Associative containers

map<K, V>	store key-value pairs of type K and V respectively ordered with respect to keys
set<K>	store a set of value of type K (no repetition) ordered with respect to keys
unordered_map<K, V>	similar to map but unordered
unordered_set<K>	similar to set but unordered

std::array<T, N>

Similar to static arrays with added benefits

T is type of elements
N is size (known at compile time)

```cpp
#include <array>
#include <iostream>
using std::cout, std::endl;

int main() {
    std::array<float, 3> data{10.0F, 100.0F, 1000.0F};

    for (const auto& elem : data)
        cout << elem << endl;

    cout << std::boolalpha;
    cout << "Array empty: " << data.empty() << endl;
    cout << "Array size : " << data.size() << endl;
}
```

`std::array<T, N>`

assuming a is initialize as:
`array<int, 5> a {1,2,3,4,5};`

Member functions (use with `.` operator)

Member function	description	example
`size_type size()`	size of the array	`cout << a.size()`
`bool empty()`	Is array empty?	`if(a.empty()) {…}`
`T front()`	first element (at index 0)	`cout << a.front()`
`T back()`	last element (at index `a.size()-1`)	`cout << a.back()`
`void fill()`	fill array with given value	`a.fill(-1)`
`T operator[]()`	access specified element (no bounds checking)	`int x = a[2]`
`T at()`	access specified element with bounds checking	`int x = a.at(2)`
`operator=()`	assignment operator	`array<int,5> b = a`

Can be passed as value to functions unlike plain arrays

std::vector<T>

Resizable arrays (dynamically allocated)

```
#include <iostream >
#include <string>
#include <vector>
using std::cout, std::endl;

int main() {
    std::vector<int> numbers = {1, 2, 3};
    std::vector<std::string > names = {"Imran", "Khan"};

    names.emplace_back("Niazi");
    cout << "First name : " << names.front() << endl;
    cout << "Last number: " << numbers.back() << endl;
}
```

std::vector<T>

All methods of arrays are also available in vectors plus more e.g.

Member functions (use with `.` operator)

assuming v is initialized as:
`vector<int> v {1,2,3,4,5};`

Member function	description	example
`void push_back()` `void emplace_back()`	append an element at the end of the vector `v.size()` will be incremented by 1	`v.push_back(6)` `v.emplace_back(7)`
`void pop_back()`	remove last element `v.size()` will be decremented by 1	`v.pop_back()`
`void clear()`	remove all elements `v.size()` will be 0	`v.clear()`
`void reserve()`	set capacity (reserve space) `v.size()` does not change	`v.reserve(100)`

Use it! It is fast and flexible!
Consider it to be a default container to store collections of items of any same type

Optimize vector resizing

`std::vector` size unknown in the beginning (v does not know how many elements will be inserted)

Therefore, a `capacity` is defined (reserved space for new elements)

`size <= capacity`

Many `push_back`/`emplace_back` operations force vector to change its capacity many times
Capacity doubled whenever need to make space for new element

`reserve(n)` ensures that the vector has enough memory to store n items
The parameter n can even be approximate

This is a very important optimization

Optimize vector resizing

```
vector<int> vec;  // size 0, capacity 0
vec.reserve(N);   // size 0, capacity 100
for (int i = 0; i < N; ++i)
    vec.emplace_back(i);
// vec ends with size 100, capacity 100
```

All insertions are fast (no resizing in for loop)

```
vector<int> vec2;   // size 0, capacity 0
for (int i = 0; i < N; ++i)
    vec2.emplace_back(i);
// vec2 ends with size 100, capacity 128
```

Some insertions are slow
Resizing happened when there is no reserved space for newly inserted element

Detour: Implementing our own vector

```
struct vec {
    int capacity = 1; // reserved space
    int size = 0;     // used space
    int *data = new int[capacity];
};
```

```
void push_back(vec& v, int e) {
    if(v.size >= v.capacity)
        increase_capacity(v);
    v.size++;
    v.data[v.size - 1] = e;
}
```

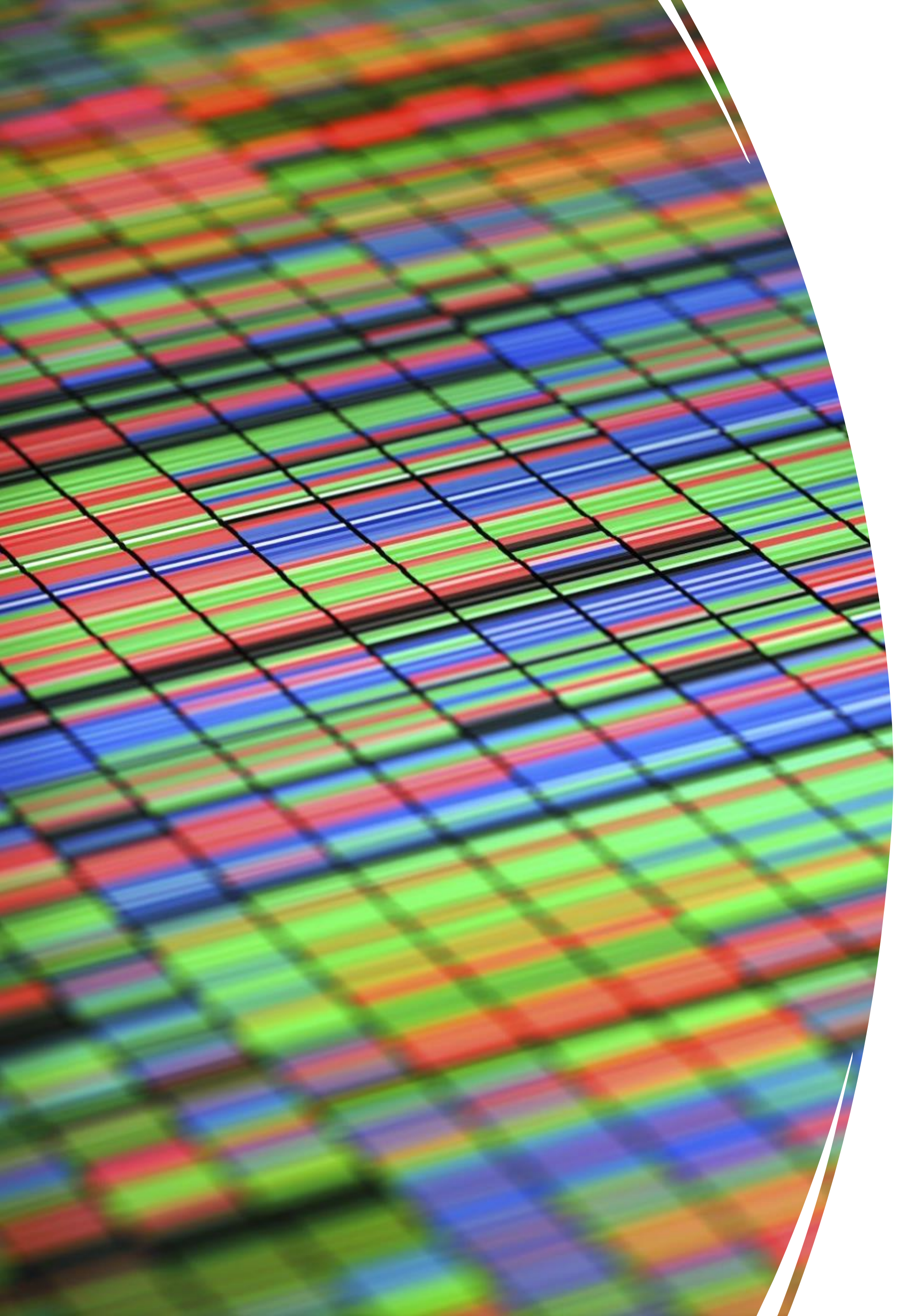
helper function, used by push_back()

```
void increase_capacity(vec& v) {
    int *newdata = new int[2*v.capacity];
    for(int i=0; i<v.capacity; i++)
        newdata[i] = v.data[i];

    delete[] v.data;
    v.data = newdata;
    v.capacity *= 2;
}
```

allocate new array with larger capacity and copy all elements from old array

test client

```
int main() {
    vec v;
    push_back(v, 9);
    push_back(v, 8);
    push_back(v, 7);
    cout << v.capacity << " \n";
    for(int i=0; i<v.size; i++)
        cout << v.data[i] << ",  ";
}
```



Abstract Data Types

Abstract Data Types

An abstract data type is a data type whose representation is hidden from the client.

Example: std::`string` or `std::vector` in standard library

- We can use it without knowing how they are represented
- Compare to complex data type implemented earlier where representation of complex numbers is separate from operations on complex values

Impact: Clients can use ADTs without knowing implementation details.

Best practice: Use abstract data types (representation is hidden from the client).

Object oriented programming

Object-oriented programming (OOP).

- Create your own abstract data types.
- Use them in your programs (manipulate objects)

Examples:

data type	set of values	example of operations
string	sequence of characters	size, substring, compare
vector	sequence of values	size, access, insert
map	key-value pairs	size, insert