NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4102 – Prosedyre- og objektorientert programmering

Faglig kontakt under eksamen: Lasse Natvig

Tlf.: 906 44 580

Eksamensdato: 16 Mai 2018

Eksamenstid (fra-til): kl. 0900 - 1300

Hjelpemiddelkode/Tillatte hjelpemidler: C: Spesifiserte trykte og håndskrevne hjelpemidler tillatt. Bestemt, enkel kalkulator tillatt. Walter Savitch, Absolute C++

Målform/språk: Bokmål

Antall sider: 10 inkludert vedlegg

Vedlegg 1: Ark om C++11 (1 side)

Kontrollert av:

Informasjon om trykking av eksamensoppgave
Originalen er:
1-sidig X **2-sidig** ☐
sort/hvit ☐ **farger X**

Dato Sign

Merk! Studenter finner sensur i Studentweb. Har du spørsmål om din sensur må du kontakte instituttet ditt. Eksamenskontoret vil ikke kunne svare på slike spørsmål.

Generell introduksjon

Les gjennom oppgavetekstene nøye. Noen av oppgavene har lengre tekst, men dette er for å gi kontekst, introduksjon og eksempler til oppgavene.

Når det står "*implementer*" eller "lag" skal du skrive en fungerende implementasjon: hvis det handler om en funksjon skal du skrive deklarasjonen med returtype og parametertype(r) og hele funksjons-kroppen.

Når det står "*deklarer*" er vi kun interessert i funksjons- eller klassedeklarasjonen. Typisk vil dette være deklarasjoner du vanligvis finner i header-filer.

Hvis det står "*forklar*" står du fritt i hvordan du svarer, men bruk enkle kodelinjer og/eller korte tekst-forklaringer og vær kort og presis.

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger du finner nødvendig.

Hver enkelt oppgave er ikke ment å være mer omfattende enn det som er beskrevet. Noen oppgaver fokuserer bare på enkeltfunksjoner og da er det utelukkende denne funksjonen som er tema. Andre oppgaver er "oppskriftsbasert" og vi spør etter funksjoner som utgjør deler i et program, eller forskjellige deler av en eller flere klasser. Du kan velge selv om du vil løse dette trinnvis, eller om du vil lage en samlet implementasjon, men sørg for at det går tydelig frem hvilke spørsmål du har svart på hvor i koden din. Husk at funksjonene du lager i en deloppgave ofte er ment å skulle brukes i andre deloppgaver.

All kode skal være i C++. Det er ikke viktig å huske helt korrekt syntaks for bibliotekfunksjoner. Oppgaven krever ikke kjennskap til andre klasser og funksjoner enn de du har blitt godt kjent med i øvingsopplegget, eller som står beskrevet i læreboka.

Det er ikke nødvendig å ha med include-statement eller vise hvordan koden skal lagres i filer.

Hele oppgavesettet er arbeidskrevende og det er ikke forventet at alle skal klare alt. Tenk strategisk i forhold til ditt nivå og dine ambisjoner!

Deloppgavene i de "tematiske" oppgavene er organisert i en logisk rekkefølge, men det betyr ikke at det er direkte sammenheng mellom vanskelighetsgrad og nummereringen av deloppgavene.

Hoveddelene av eksamensoppgaven teller i utgangspunktet med den andelen som er angitt i prosent. Den prosentvise uttellingen for hver oppgave kan likevel bli justert ved sensur basert på hvordan oppgavene har fungert. De enkelte deloppgaver kan også bli tillagt forskjellig vekt.

Oppgave 1: Kodeforståelse (20 %)

Svar på denne måten på et vanlig svarark:
1a) … ditt svar her
1b) … ditt svar her … osv.

1a) Hva skrives ut ?

```
int i = 1;
while ((i / 32) == 0) {
        i += 2;
}
cout << i << endl;
```

1b) Hva skrives ut ?

```
int a = 20;
float b = a++;
float ab = a / b;
cout << a << " " << b << " " << ab << endl;
```

1c) Funksjonen f er definert i del (i), hva skrives ut av koden i del (ii)?

(i)

```
int f(int h, int *e, int &i) {
    h++;
    *e -= h;
    i += h;
    return h;
}
```

(ii)

```
int h = 19;
int e = 21;
int i = -12;
h = f(h, &e, i);
cout << h << e << i << endl;
```

1d) Hva skrives ut ?

```
vector<int> norge = { 14, 14, 11 };
vector<string> m = { "G", "S", "B" };
for (int i = 0; i < norge.size(); i++) {
        cout << norge[i] << " " << m[i] << "ull ";
}
cout << endl;
```

1e) Gitt definisjonene av `decs` i del (i), hva skrives ut av koden i del (ii)?

(i)

```
int decs(int v) {
        return 0;
}
int decs(float v) {
        int a = v;
        int b = (v - a) * 100;
        return b;
}
```

(ii)

```
float e = 2.71828;
cout << decs(e) << endl;
```

1f) Hva skrives ut ?

```
vector<int> v;
v.push_back(2);
v.push_back(1);
for (int i = 1; i < 4; i++) {
        v.push_back(v[i - 1] + v[i]);
}
for (auto n : v) {
        cout << n << " ";
}
cout << endl;
```

1g) Hva skrives ut ?

```
template<class T>
string thingout(T thing, int a) {
	stringstream ss;
	for (int i = 0; i < a; i++)
		ss << thing;
	return ss.str();
}
void main() {
	cout << "s:" << thingout<string>("hi", 4) << endl;
	cout << "i:" << thingout<int>(5, 3) << endl;
}
```

1h) Hva skrives ut ?

```
	int a[] = { 1,4,3,2 };
	int *p = a;
	p++;
	cout << *p << *(p + 2) << endl;
```

1i) Hva skrives ut ?

```
	int foo = 0x20;
	try {
		if (foo % 0x10 != 0) {
			throw foo;
		}
		cout << foo << " is nice!" << endl;
	}
	catch (int e) {
		cout << e << " was sad :(" << endl;
	}
```

1j) Hva skrives ut?

```
void overview(map<string, int>::iterator it,
		map<string, int>::iterator end) {
	if (it == end)
		return;
	cout << it->first << " tok " << it->second << endl;
	overview(++it, end);
}
void main() {
	map<string, int> medaljonga;
	medaljonga["Bjørgen"] = 15;
	medaljonga["Sundby"] = 3;
	medaljonga["Klæbo"] = 3;
	medaljonga["Jansrud"] = 2;
	overview(medaljonga.begin(), medaljonga.end());
}
```

1k) Hva skrives ut?

```
class Drink {
public:
	virtual string drink() {return "glug";}
};
class Coffee : public Drink {
public:
	string drink() {return "slurp";}
};
void main() {
	Coffee mug;
	Drink *glass = &mug;
	cout << mug.drink() << " " << glass->drink() << endl;
}
```

Oppgave 2: Kalender (30%)

Du har tatt på deg et viktig verv i NTNUI og har fått mye artige aktiviteter å holde rede på. For at det ikke skal gå ut over assistentjobben din i TDT4102 så har du begynt å snekre på et C++ program for en kalender. Du kan anta at programmet ditt har en tabell `months` over antall dager i hver måned:

```
const int months[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

For skuddår får vi en ekstra dag i februar, altså 29 dager istedenfor 28. Skuddår er normalt hvert 4. år (…, 2016, 2020, 2024, …). Unntaket er hundreårene (…, 1700, 1800, 1900, …) som ikke er skuddår med mindre de er delelige på 400 (…, 1600, 2000, 2400 …).

2a) Skriv en funksjon **`bool isLeapYear(int year)`** som returnerer `true` hvis `year` er skuddår og `false` ellers.

2b) Skriv en funksjon **`int daysInMonth(int m, int y)`** som tar inn måned `m` (1 - 12) og år `y` som parametre og returnerer antall dager i den aktuelle måneden. Husk skuddår, som beskrevet ovenfor.

2c) Det er deklarert en datatype `Date` som i Figur 1. Skriv en funksjon **`int dayNo(Date day)`** som returnerer dag-nummer i år, slik at 1 Januar er dag nr. 1, 2 februar er dag nr. 32 osv. Hint: Bruk **`daysInMonth()`** fra forrige deloppgave.

```
struct Date {
        int d; // day
        int m; // month
        int y; // year
};
```

Figur 1.

2d) I programmet har vi behov for å lese inn datoer fra fil. I den forbindelse trenger vi en funksjon som konverterer en dato representert som tekst (**`string`**) til et `Date`-objekt. Tekststrengen har formatet "`YYYY-MM-DD`", der `YYYY` er året, `MM` er et heltall mellom 01 og 12 og `DD` er dagen i den aktuelle måneden `MM`. *Merk at året skrives alltid med fire tegn mens både måned og dag skrives med to tegn, f.eks. 4. mars i år 850 blir* `0850-03-04`.

Skriv en funksjon **`stringToDate()`** som tar som parameter en tekststreng og returnerer et Date-objekt med riktig innhold i til tekststrengen som beskrevet over. Om tekststrengen er på feil format skal det kastes et unntak (Eng. exception). Du trenger ikke sjekke om verdiene for år, dag og måned er gyldig.

Hint: her kan det være nyttig å bruke funksjonen `stoi()` som forsøker å tolke strengen `str` som et heltall og returnerer tallet. `stoi()` kaster unntaket `std::invalid_argument` om strengen ikke kan tolkes som heltall (f.eks. "p2").

2e) Skriv en funksjon **`bool checkDate(Date day)`** som sjekker om datoen `day` er en gyldig dato. En dato er gyldig om måneden er innenfor området 1-12 og dagen er mellom 1 og antall dager i måneden (husk skuddår). Funksjonen skal kun returnerer true om datoen er gyldig.

2f) Skriv en funksjon `string` **`dateToString`**`(const Date& day)` som tar inn et Date-objekt og returnerer en tekststreng på formatet "`YYYY-MM-DD`" som beskrevet i deloppgave 2d). Om datoen ikke er en gyldig dato, skal funksjonen returnere tekststrengen "`INVALID DATE`".

2g) Deklarer en ny datatype `struct` **`Event`** som representerer en hendelse. Denne skal ha de tre medlemsvariablene `name` av type **`string`**, `id` av type `int` og `when` av type `Date`. Implementer `std::ostream & operator<<(std::ostream & out, const Event & rhs)` som skriver til `out` innholdet til Event-objektet på formatet "`id : name @ YYYY-MM-DD`", f.eks. "`212 : Important meeting @ 2018-03-20`".

2h) Skriv en funksjon `void printEvents(vector<Event *> &events)` som skriver ut alle hendelser lagret i vektoren på formatet spesifisert i deloppgave 2g. Merk at events er en referanse til en vector av *pekere*. Det skal være én hendelse per linje, og til slutt skal det skrives ut én linje med teksten "*n* events", der *n* er antallet hendelser i vektoren.

```
class Calendar {
private:
	string name; // name of calendar
	map<int, Event *> events; // map of events indexed by id
public:
	Calendar(string name) : name(name) {}
	~Calendar();
	void addEvent(int id, string name, int year, int month, int day);
	vector<Event *> getEvents(int year, int month, int day);
	vector<Date> busyDates(int threshold);
};
```

Figur 2.

2i) Vi deklarerer klassen `Calendar` som vist i Figur 2. Implementer `Calendar::addEvent()` som dynamisk allokerer et nytt `Event`-objekt med innholdet gitt i parameterlisten, og legger den til i `events`. Om en hendelse med samme id allerede eksisterer, eller om datoen ikke er gyldig, skal det kastes et unntak.

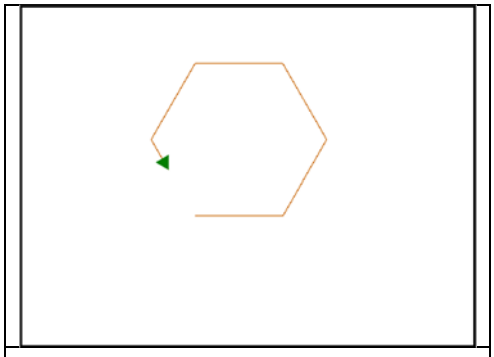
2j) Implementer destruktøren til `Calendar`.

2k) Implementer `Calendar::getEvents()` som skal returnere en vektor med alle hendelser som faller på dagen gitt av parameterlisten.

2l) Funksjonen `Calendar::busyDates(int threshold)` teller antall hendelser som skjer på en og samme dato. Den returnerer så en vektor med de datoene som har minst `threshold` antall hendelser. Implementer `Calendar::busyDates()`. Du kan anta at `operator<` er implementert for Date.

Oppgave 3: Turtle graphics (30%)

Turtle graphics er et rammeverk for å tegne vektorgrafikk. Rammeverket tar utgangspunkt i en skilpadde (Eng. turtle) som befinner seg på et 2D lerret (Eng. canvas). Skilpadden har en posisjon og en retning, kan bevege seg fremover eller bakover i en rett linje, eller snu seg mot venstre eller høyre. Når skilpadden beveger seg, tegner den en linje på lerretet som vist i Figur 3. I denne oppgaven skal du implementere deler av en klasse for Turtle graphics ved hjelp av grafikkrammeverket SFML som du har blitt kjent med gjennom øvingsopplegget.



Figur 3.

Farger i SFML (og mange andre tilsvarende systemer) er basert på RGB, dvs. en blanding av de tre primærfargene rød, grønn og blå. Hver av de tre farge-komponentene i RGB er representert som et heltall mellom 0 og 255. I SFML har vi datastrukturen `sf::Color` som representerer en farge. Som et eksempel så vil `sf::Color myViolet = sf::Color(127, 0, 127);` opprette et objekt `myViolet` som er en blanding av rødt og blått, dvs. fiolett. SFML har også predefinerte farger slik som f.eks. `sf::Color::White`.

Et *vektor-objekt* i SFML (`sf::Vector2f`) er som en vektor i matematikk (geometri) altså et punkt i x-y-planet. I SFML er konstruktøren for en vektor deklarert som `sf::Vector2f(float x, float y);` Merk at vi bruker flyttall (float) for posisjoner i SFML, selv om størrelsen på tegneområdet kun har et gitt antall piksler.

Klassedeklarasjonen for Turtle er vist nedenfor. Vi vil forklare de ulike medlemsvariablene og funksjonene etter hvert som de er aktuelle for deloppgavene.

```
class Turtle {
protected:
	float x, y; // Current position (x,y) relative to center of canvas
	float angle; // Current angle (degrees)
	int width;    // Size of canvas in pixels:
	int height;   // width in x-direction, height in y-direction
	float lineWidth; // Line width of pen
	sf::Color color; // Line color
	sf::Color bgColor; // Background color
	sf::RenderTexture canvas;  // Drawing canvas
	// drawLine will draw line from (x0, y0) to (x1, y1) on the canvas,
	// with lineWidth as the width of the line and color as the line color.
	// The origin (0.0, 0.0) is defined as center of the canvas
	void drawLine(float x0, float y0, float x1, float y1);
public:
	Turtle(int width, int height, sf::Color bgColor = sf::Color::White,
		sf::Color color = sf::Color::Black);
	void clear();  // Fill canvas with bgColor
	sf::Vector2f getPosition();
	void setPosition(float x, float y);
	float getAngle() { return angle; }
	void setAngle(float angle) { this->angle = angle; }
	int getWidth() { return width; }
	int getHeight() { return height; }
	sf::Color getColor() { return color; }
	void setColor(sf::Color color) { this->color = color; }
	void setLineWidth(float lw) { lineWidth = lw; }
	virtual void forward(float distance);
	virtual void backward(float distance);
	virtual void left(float angle);
	virtual void right(float angle);
};
```

3a) Variablene `x` og `y` utgjør posisjonen til skilpadden på tegneområdet (`canvas`). Posisjonen (0, 0) er definert som midten av tegneområdet, hvor positiv x-retning er mot høyre mens positiv y-retning er oppover. Implementer medlemsfunksjonene `Turtle::setPosition` og `Turtle::getPosition` som setter og returnerer posisjonen (x, y) i henhold til klassedeklarasjonen. Du trenger ikke sjekke om posisjonen er innenfor tegneområdet.

3b) Implementer konstruktøren til `Turtle`. Initialiser *alle* medlemsvariabler. `x`, `y` og `angle` skal initialiseres til 0.0 mens `lineWidth` skal settes til 1.0. Variabelen `canvas` er et lerret (canvas) som SFML gir oss for å tegne på. Tegneområdet består av `width` antall bildelementer (piksel) i bredden, og `height` antall piksel i høyde-retning. For å initialisere tegneområdet må konstruktøren kalle `canvas.create(width, height)`. Merk at `canvas.create()` kan feile - den vil da returnere false. Om

det skjer skal det kastes et unntak. Til slutt skal konstruktøren kalle `Turtle::clear()` for å fylle tegneområdet med bakgrunnsfargen.

3c) Funksjonen `forward(float distance)` flytter skilpadden framover `distance` antall piksler og tegner en linje på veien. Retningen er gitt av verdien på `angle` i grader. Verdien 0.0 tilsvarer rett mot høyre langs x-aksen, verdien 90 tilsvarer rett opp, 180 rett mot venstre osv. Derfor vil den nye posisjonen (x1, y1) være gitt av:

x1 = x0 + `distance` * cos(`angle`) og y1 = y0 + `distance` * sin(`angle`)

når vi antar at gjeldende posisjon er (x0, y0). Implementer funksjonen `void Turtle::forward(float distance)`. Bruk funksjonen `float deg2rad(float deg)` for å konvertere fra grader (`deg`) til radianer. Bruk funksjonen `Turtle::drawLine()` som er forklart i klasse-deklarasjonen for å tegne linjen. Du trenger ikke ta hensyn til om linjen havner utenfor `canvas`, da SFML håndterer dette for oss automatisk.

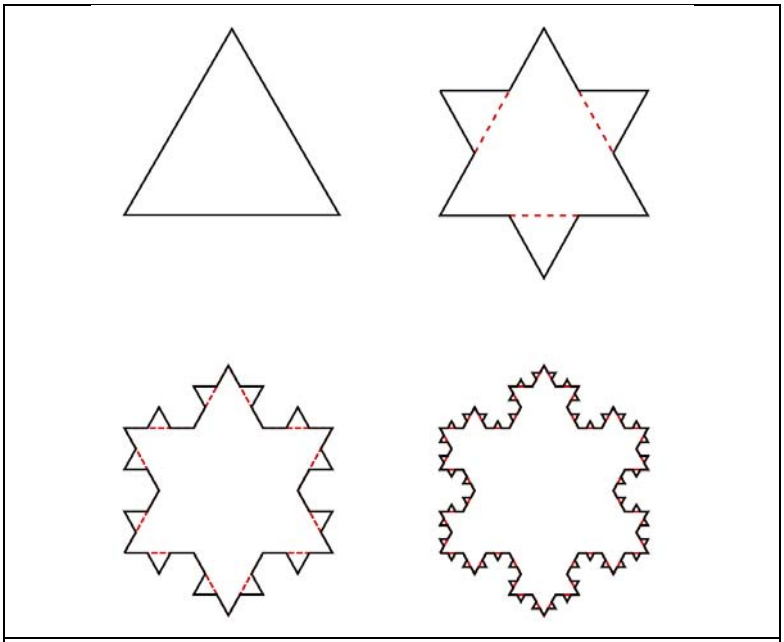
3d) Implementer funksjonen `backward()` som flytter skilpadden bakover `distance` antall piksler. (Hint: du kan implementere `backward` ved hjelp av `forward`).

3e) Implementer funksjonen `left()` som snur skilpadden angle antall *grader* mot venstre. Implementer også `right()` som snur den mot høyre.

3f) Vi skal nå ta i bruk Turtle-klassen for å tegne. Implementer en funksjon `void hexagon(Turtle *turtle)` som tar inn en Turtle-peker og bruker den til å tegne en sekskant. (Slik som Figur 3, men helt «fullført» ved at skilpadden når frem til utgangspunktet). Sekskanten skal ha fargen RGB(200, 100, 0) og sidene skal være 100 piksler lange.

3g) Vi skal nå bruke Turtle-klassen til å simulere en såkalt «random walk». Skriv en funksjon `float randRange(float min, float max)` som returnerer et tilfeldig tall i området [`min`, `max`]. Bruk så denne til å implementere funksjonen `void randomWalk(Turtle *turtle, int steps)` som flytter skilpadden `steps` antall ganger (skritt). For hvert skritt skal skilpadden endre retning med en tilfeldig vinkel mellom 90 og -90 grader og flytte forover et tilfeldig antall piksler mellom 0 og 10.

3h) Kochs snøflak er et eksempel på en såkalt matematisk fraktal (Se Figur 4). For å generere en slik figur starter vi med en likesidet trekant (nivå 0, øverst til venstre i figuren). Nivå 1 får vi ved å dele opp alle sidene i 3 like segmenter, og erstatter det midterste segmentet med 2 nye linjer som danner en ny mindre likesidet trekant (øverst til høyre i figuren). Det er bare den sorte streken i figuren som skal tegnes, den røde er bare med for å illustrere rekursjonen. Om vi fortsetter denne prosedyren for alle segmentene får vi de to neste nivå



Figur 4.

som vist nederst i figuren. Turtle graphics kan brukes til å generere Kochs snøflak på en elegant måte ved hjelp av rekursjon.

Først skal vi skrive kode for å generere *én side* av fraktalen. Implementer funksjonen

```
void koch(Turtle *turtle, float length, int level)
```

som genererer én side av fraktalen for et gitt nivå `level`. Lengden på den første linjen skal være `length`. Hint: Bruk rekursjon, reduser lengden med en tredel for hvert nivå, og flytt skilpadden forover kun når nivå 0 er nådd. Her kan det være greit å fokusere på den *nederste linjen* i trekanten på nivå 0 for å se hvilke bevegelser som skal utføres for å komme til nivå 1.

3i) Vi skal nå ta i bruk funksjonen `koch`() for å tegne hele snøflaket. Implementer funksjonen `void snowflake(Turtle *turtle, float length, int level)` som tegner alle de 3 sidene av snøflaket for et gitt nivå `level`. Start med den nederste siden av fraktalen. Du kan anta at skilpaddens posisjon er i midten av lerretet og har retning mot høyre.

Oppgave 4: Roboten Wally (20 %)

I denne oppgaven skal du bruke `Turtle`-klassen fra oppgaven ovenfor til å implementere en enkel robotsimulator. Du trenger ikke ha løst eller forstått alt i oppgave 3 for å kunne løse oppgave 4, men du må sette deg inn i klasse-deklarasjonen og beskrivelsen av noen av medlemsfunksjonene.

4a) Deklarer en klasse `Wally` som er en spesialisering av klassen `Turtle` og kodet ved å bruke arv. Klassen skal ha to nye private medlemsvariabler av typen float: `speed` (hastighet forover målt i antall piksel per skritt) og turn («snuhastighet», dvs. antall grader endring av retningen (angle) per skritt). Deklarer og implementer inline get- og set-funksjoner for disse to medlemsvariablene.

4b) Deklarer og implementer konstruktøren. Den skal ta som parametere `width` og `height` samt `speed` og `turn` som verdier for de nye medlemsvariablene.

4c) Roboten befinner seg i et rom med størrelse angitt av `Turtle::width` og `Turtle::height`. Vi ønsker å modifisere `forward`() slik at den ikke kan bevege seg utenfor dette rommet. Implementer `Wally`::`forward`() som flytter roboten fremover, og sørger for at roboten ikke beveger seg utenfor rommet. Velg en enkel løsning. Du trenger ikke beregne krysningspunktet til veggen. Husk at (0,0) er midten av rommet.

4d) Du skal nå skrive en funksjon `Wally::run`() som styrer roboten. Roboten er utstyrt med en avstandssensor som måler avstanden foran seg til nærmeste vegg. Anta at funksjonen `float Wally::sense()` allerede er implementert, som returnerer denne avstanden. Roboten Wally beveger seg fremover og snur når den nærmer seg en vegg. Den måler kontinuerlig avstanden til veggen foran seg:

- Om avstanden er mindre enn 100: snu mot venstre `turn` antall grader
- Om avstanden er større enn 10: flytt fremover `speed` antall piksler

Ingen robot er perfekt, og det vil være knyttet usikkerhet til all bevegelse. Simuler dette ved å snu Wally et tilfeldig antall grader mellom -5 og 5 *hver* gang den flytter på seg.

Implementer `run`() som styrer roboten som beskrevet over. Roboten skal gå uendelig lenge (evig løkke).

…---oooOooo---…

Initializer lists

```
/* Initialize vec using initializer list */
vector<string> vec = { "AB", "CD", "EF" };
```

auto

```
auto it = vec.begin();
const auto& first = vec[0];
```

Range-based for-loops

```
for (string str : vec)
  cout << str << endl;
/* Avoid copy */
for (const string& str : vec)
  cout << str << endl;
/* Avoid copy and use auto */
for (const auto& str : vec)
  cout << str << endl;
```

Enum class (strongly typed enums)

```
enum class Color : int {red, green, blue};
enum class Feeling : int {sad, happy, blue};
/* Need explicit cast to convert to int */
int x = static_cast<int>(Color::blue);
/* Feelings and Colors can't be compared */
if (Color::blue == Feeling::blue) {
  /* compile error */
}
```

nullptr

```
/* nullptr indicates null pointer value */
int* ptr = nullptr;
```

to_string

```
to_string(23.45); /* -> "23.450000" */
to_string(3) /* -> "3" */
```

Delegating constructors

```
class Student {
public:
  Student(string username, string email);
  Student(string username)
   : Student(username,
             username + "@stud.ntnu.no")
  {}
};
```

Default member initialization

```
/* Applies also to struct */
class Person {
  bool alive = true;
  /* ... more code ... */
};
```

std::array

```
/* Array of 5 ints */
array<int, 5> arr = {1,2,3,4,5};
/* size() member function */
cout << arr.size() << endl;
/* operator[] is used to access elements */
cout << arr[0] << endl;
/* can be used with iterators and STL */
sort(arr.begin(), arr.end());
for (int elm : arr)
   cout << elm << endl;
```

Unique pointer

`std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` is destructed. When using the C++14 function `std::make_unique`, there is no need to manually allocate memory for the object. The `unique_ptr` *owns* the object - it does not allow the pointer to be copied, but the ownership may be *transferred* using `std::move`. It is possible to check whether an `unique_ptr` manages an object by comparing it to `nullptr`. The `unique_ptr` can be used just like a normal pointer, using the indirection operator (`*`) or the arrow operator (`->`).

```
/* create unique_ptr using make_unique - the preferred way */
unique_ptr<Student> s1 = make_unique<Student>("daso");
/* create unique ptr using unique_ptr's auto */
auto s2 = make_unique<Student>("lana");
/* transfer ownership of unique ptr */
auto s3 = move(s1); /* value of s1 is now unspecified */
cout << s3->getName() << " " << (*s3).getEmail() << endl;
```

In cases where we want a function to use the object pointed to **without** transferring ownership to the function, we can get the underlying raw pointer.

```
void printStudent(Student* sPtr);
printStudent(s2.get()); /* s2.get() returns the underlying raw pointer */
```