6. Intro till avancerad C++ (Tobias grupp)

18 juli

Syftet med dagens lektion är att lära er så många coola saker om C++ som möjligt, under antagandet att ni redan har lite kunskap om språket. Hoppas ni gillar det :). C++ är ett språk som har massor med features. Många bra, och en del som är gamla och kanske bör undvikas. Vi kommer under dagens lektion bara att gå igenom features som vi tycker är intressanta/viktiga för att kunna programmera i C++, eller för att förstå saker bättre.

Uppgifterna här är uppdelade i olika delar. Ifall ni känner att ni redan vet massor om ett visst ämne så kan ni hoppa över det ämnet, men läs gärna igenom uppgifterna först för att se ifall det finns något ni inte kan. Vi har försökt att ge en hel del exempel, men ifall ni vill ha mer info om det som vi inte tar upp så är det bara att fråga eller googla er fram till mer information då det är omöjligt för oss att ta upp allting.

Att köra ett program

För att köra ett program skrivet i C++ måste man först kompilera det. Det kan man göra genom att skriva `g++ file.cpp -o output_name` i terminalen. Sedan kan man köra programmet genom att skriva `./output_name`.

Kompileringsflaggor

Det är väldigt vanligt att program kraschar, oavsett vilket språk man kodar i. Något som är svårt i C++ jämfört med en del andra språk är att förstå vad som har gått fel. Därför ska vi titta på ett exempelprogram som kraschar för att se ifall vi kan förstå det bättre.

1. För följande exempel, identifiera först vad koden gör och vad som kan göra att programmen kraschar. Kör sedan programmen och se vad ni får för output när programmet kraschar.

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {10,20,50};
    int n;
    std::cin >> n;
    for(int i = 1; i <= n; i++){
        std::cout << v[i] << std::endl;
    }
    return 0;
}
```

Exakt vad ni får för output kan variera beroende på bland annat vilken dator ni kör på och vilka kompileringsflaggor ni använder. Kompilerade ni programmet genom att skriva `g++ program.cpp -o output_name`, så fick ni antagligen en rad output där det står: Segmentation fault (core dumped)

Det säger inte riktigt vad som orsakade problem, och kan generellt sett orsakas på många olika sätt. Ett sätt som vi kan minska risken att få dessa kryptiska meddelanden är att lägga till en kompileringsflagga: `-fsanitize=address`. Kompilera om programmet med samma kommando som innan, men lägg till den här flaggan. Om ni kör om programmet nu så lär ni se att ni får mycket mer text som beskriver vad som gick fel.

Bland annat finns en stack trace av var kraschen skedde. Men var nånstans i koden skedde kraschen? Det står med, men är väldigt kryptiskt. Vi kan lägga till ytterligare en kompileringsflagga för att se till så att det blir lättare att läsa det: `-g`. Lägg till `-g` och kompilera om igen. Nu bör ni se ett par radnummer som del av outputen. Det finns även en hel del annan information som är bra att förstå, men det är lite för mycket för att beskriva allt här. Fråga en ledare om hjälp så berättar de mer!

Vi rekommenderar att ni använder dessa flaggor varje gång ni testkör era C++ program. Men finns det någon nackdel med detta? Vad tror ni?

2. Testkör följande kod. Vad får ni för output. Försök först att komma fram till vad det borde bli och kör det sedan.

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 128;
    char c = x;
    cout << (int)c << endl;
}
```

3. Lägg till kompileringsflaggan `-Wconversion` och kompilera om programmet från förra uppgiften. Vad ser ni nu?

Som ni ser så kan man kontrollera vilka varningar som genereras när man kompilerar sin kod. Det är väldigt användbart att få dessa varningar då det signalerar att det kanske finns en bugg i ditt program. Det finns alldeles för många kompileringsflaggor för att gå igenom alla här. Men ett par vanliga och användbara flaggor är:

- `-Wall` - Står för warning all, dock sätter den faktiskt inte på alla varningar. Men den varnar för mycket saker.
- `-Werror` - Står för warning error. Det gör att alla varningar man får blir till errors istället, vilket gör att kompileringen failar. Tanken med detta är att tvinga programmerare att skriva kod som inte generar varningar, och på så sätt undvika buggar.

- `-O0` - Optimeringsnivå 0. Det finns fortfarande ett fåtal optimeringar som är påslagna, men det mesta är av. Detta är bra för debugging då koden gör i princip exakt det du skrev, rad för rad.
- `-O1, -O2, -O3` - Olika optimeringsnivåer. Ju högre tal desto mer opitemerar kompilatorn din kod, vilket leder till snabbare exekverbara filer.

Typer/storlekar av typer

I C++ måste man för det mesta vara explicit med typerna. Det finns massor av typer att välja mellan, och vad man vill använda beror på syftet. T.ex. för heltal så räcker det ofta med en int, så man behöver inte använde mer än så då. Men behöver man spara större heltal så kan man byta typ. Ett par av de vanligaste typerna som finns inbyggda är:

```
// Tekniskt sett så har följande typer inte samma storlek på alla system. Men
    på i princip alla system så gäller följande storlekar.
char // Heltalsvärde med 8 bitar.
short // Heltalsvärde med 16 bitar.
int // Heltalsvärde med 32 bitar.
long long // Heltalsvärde med 64 bitar.

float // Decimaltal som sparas som 32 bitar.
double // (Bättre än floats) Decimaltal. Sparas som 64 bitar i minnet.
long double // Decimaltal. Sparas som 80 bitar i minnet, men kanske inte finns
    på typ vissa windowssystem.

unsigned char
unsigned short
unsigned int
... // Att lägga till unsigned gör att det går att representera större
    positiva tal då det inte längre går att lagra negativa tal i datatypen.
    Dock är det ibland farligt att använda unsigned datatyper då det är lätt
    att råka ut för underflow.

char * // Tolkas ofta som en sträng, särskilt i C.

// Man kan även skapa arrays på stacken genom att skriva någonting i stil med:
int arr[5]; // Innehåller 5 ints.

// Man kan beräkna antalet bytes av en variabel/datatyp genom att skriva t.ex.:
sizeof(int) // == 4. För typer krävs en parantes.
sizeof arr // == 20. För variabler krävs det inte en parantes.
```

4. Vad händer ifall du skriver ut sizeof på en pekare?
5. Hur kan du beräkna längden på en array?

Man kan även använda auto ibland för att skippa att skriva ut typen. Då skriver man `auto x = ...;` och kompilatorn kommer automatiskt att lista ut vilken typ x ska ha. Detta används ibland, men bör inte överanvändas då det är bra att lätt

kunna se vilken typ en variabel har. Det är dock användbart när man t.ex. skapar variabler som har typer som är väldigt långa att skriva, och som bara kommer att användas i ett väldigt litet scope.

Includes/använda flera filer

Ifall man vill använda sig utav funktioner som ligger i en annan fil så måste man inkludera en fil som innehåller en deklaration av den funktionen. Detta gör man genom att skriva t.ex.:

```
#include <iostream> // For things like cout, cerr etc.
#include "my_other_cool_file.hpp"
```

Man kan alltså använda antingen <> eller "" men enligt kompilatorn så betyder det samma sak. Det är dock ganska standard att man brukar använda <> för standardbiblioteket, och "" för egna filer.

Det som `include` egentligen gör är att är bara att copy pasta hela filens innehåll så att det finns tillgängligt. Därför skulle man kunna tro att saker lätt kan gå fel ifall man råkar inkludera samma fil många gånger. Men det kan man lätt lösa genom att skrive `#pragma once` i början av en fil. Det ser till så att en fil bara inkluderas första gången den försöker bli inkluderad.

Datastrukturer

Ni är förmodligen bekanta med en del datastrukturer redan, men här kommer ett par korta uppgifter där ni kan lära er att använda de vanligaste datastrukturerna.

T.ex.:

```
#include <vector>
#include <unordered_map>
#include <iostream>
using namespace std;
int main()
{
    vector<int> my_cool_vector; // Creates an empty vector.
    my_cool_vector.push_back(5);
    cout << my_cool_vector[0] << endl;

    unordered_map<int, int> my_cool_map; // Creates an empty unordered map.
    my_cool_map[5] = 7;
    my_cool_map[10] = 20;
    cout << my_cool_vector[10] << endl;
    coutt << my_cool_vector.count(8) << endl;
}
```

6. Skriv ett program där ni skapar en `vector` som innehåller strings. Skriv sedan en funktion som tar bort de strängar som innehåller bokstaven `'a'`.

7. Skriv en funktion som returnerar hur många unika element det finns i en

`vector` av integers. Tips: för en bra tidskomplexitet, använd er utav antingen `std::set` eller `std::unordered_set`.

8. Skriv kod som loopar igenom en unordered_map, och skriver ut dess inehåll, genom att använda auto.

Mer avancerade datastrukturer

Det finns även många andra väldigt användbara datastrukturer som finns inbyggda i standardbiblioteket. Ett par exempel är: `std::map`, `std::unordered_map`, `std::queue`, `std::priority_queue`, `std::stack`, `std::bitset`, `std::pair`, `std::tuple`.

9. Använd en tuple för att returnera en sträng, en int och en float samtidigt från en funktion.

Pass by reference/pointer/value

När man skriver en funktion i C++, så väljer man vilka typer som argumenten har. Det finns tre huvudsakliga alternativ för detta, som vi kommer att gå igenom nu.

Det enda som ändrar på vilket sätt man skickar nått till en funktion är hur funktionen är deklarerad.

```
// Option 1:
// Calling this function will copy the whole vector, which can sometimes slow
    things down.
int some_cool_function(std::vector<int> vec);

// Option 2:
// To call this function you need to provide the pointer to the vector. It
    won't be copied.
int some_cool_function(std::vectior<int> *vec);

// Option 3:
// You call this function in the same way as option 1, but in this case no
    copy of the original vector is created. Instead, it is passed as a
    reference. This means that if you modify the vector in the function, the
    vector that was used to call the function is also modified.
int some_cool_function(std::vector<int> &vec);
```

Iteratorer

I C++ är det ganska vanligt att använda sig av iteratorer. Detta är i princip en klass som tillåter att man lätt kan stega igenom data på ett sekventiellt sätt. Till exempel ifall man vill loopa igenom en `map` så använder kan man sig utav en iterator:

```
map<string, string> var;
auto iter = begin(var);
while(iter != end(var)){
```

```
    string key = iter->first;
    string value = iter->second;
    iter++;
}
```

Iteratorer användes även väldigt ofta utan att man ens tänker på det. T.ex. som input till en del funktioner som vi kommer ta upp senare. Notera att pilsyntaxen är som att skriva en punkt på en variabel, fast man använder det för pekare och iteratorer.

10. Skapa en iterator som går igenom en vector. Stega igenom listan på ett sånt sätt att du printar var femte element i listan.

Lambdas

Lambdafunktioner är väldigt lika vanliga funktioner, förutom att de inte har något namn. Det gör det väldigt lätt att skicka det som input till en funktion som tar in en funktion som input, men man kan även spara det i ev variabel. Här är ett exempel på hur man skapar en lambdafunktion som tar in en int och en string, och returnerar en int:

```
...
int main()
{
    auto cool_lambda_function = [](int x, std::string s){
        int return_value = ...;
        return return_value;
    };
}
```

Mycket här ser ut som en vanlig funktion förutom att det finns ett semikolon på slutet och att det finns ett mystiskt `[]` efter likhetstecknet. I den kan man specifiera vilka variabler lambdafunktionen ska komma åt från det yttre scopet. Just så som den är skriven här kan funktionen inte komma åt några variabler från main funktionen. Vill man att den ska göra det så går det att specifiera. Man kan till och med välja exakt vilka variabler lambdafunktionen får ha access till, och ifall det ska vara by value eller by reference. Ifall man vill att lambdafunktionen ska ha access till alla funktioner by reference i main funktionen så skriver man:

```
...
int main()
{
    auto cool_lambda_function = [&](int x, std::string s){
        int return_value = ...;
        return return_value;
    };
}
```

11. Skriv en lambdafunktion som kvadrera ett heltal.

12. Skriv en lambdafunktion som lägger till ett tal 5 gånger i en vector som är definierad i den yttre funktionen, utan att skicka in det som en parameter.

Standardbiblioteket och ett par användbara funktioner

Det finns massor av användbara funktioner i standardbiblioteket. Här kommer ett par av dessa:

```
#include <vector>
#include <algorithm>
int main()
{
    vector<int> x = ...;
    sort(begin(x), end(x)); // Takes an iterator to the first element and an
        iterator to the end of the thing that you are sorting. This allows you
        to do things like easily sorting part of the vector etc.
}
```

13. Sortfunktionen kan även ta in en tredje parameter: en funktion som avgör hur man ska sortera innehållet. Detta kan antingen vara en funktion som du har definierat på ett annat ställe i ditt program, eller en lambdafunktion.

Skriv kod som sorterar en vector av ints med hjälp av en lambdafunktion så att den sorteras i stigande ordning på absolutbeloppet av innehållet. Om du bara ska använda lambdafunktionen på ett ställe så behöver du inte mellanlagra den i en variabel.

14. Det finns en funktion som heter `reverse`. Den tar in två iteratorer precis som sort. Skriv ett program som vänder på en vector med hjälp av denna funktionen.

Ett par andra väldigt användbara funktioner är `lower_bound` och `upper_bound`. Dessa funktioner utför en binärsökning för att hitta det man letar efter i ett intervall. Eftersom det är en binärsökning så krävs det att datan är sorterad. Både `lower_bound` och `upper_bound` tar in tre argument: först två stycken iteratorer precis som med de tidigare funktionerna, och sedan det värde man söker efter.

15. Experimentera med att använda `lower_bound` och `upper_bound` för att binärsöka genom en sorterad lista. Försök att se ifall ni kan avgöra vad skillnaden är mellan dessa funktioner. Notera att returvärdet är en iterator till det värde som binärsökningen hittar.

Pekare

En pekare är en variable som är lika med en minnesadress. Det används överallt i C kod, och även en del i C++. Man kan använda pekare till massor av saker. T.ex. kan man skapa en array genom att använda pekare, eller så kan man skicka pekare till en funktion för att få något som liknar en referens.

```
int *arr = new arr[10]; // Creates a pointer that points to memory containing
    10 ints.
...
```

```
delete arr[]; // You need to do this in order to free memory. Otherwise you
    won't be able to reuse this memory.
```

16. Skapa ett program där du inuti en while-loop allokerar små arrayer utan att köra delete. Vad händer ifall du låter programmet köra i ett par sekunder/minuter?

Smart pointers

Det finns ett par problem med vanliga pekare. Dels så måste man alltid frigöra det minne som man allokerar, vilket är väldigt lätt att glömma. Sen ifall det finns flera delare av programmet som pekar på samma minne så är det svårt att veta vilket del av programmet som blir ansvarig för att frigöra minnet. Därför är det ofta bra att undvika vanliga pekare ifall man kan. Istället kan man använda smarta pekare:

```
#include <memory>
void f(std::shared_ptr<some_class> ptr){}

int main()
{
    std::shared_ptr<some_class> ptr1 = std::make_shared<some_class>();
    std::shared_ptr<some_class> ptr2 = ptr1;
    f(ptr2);
}
```

Shared pointers tillåter att det kan finnas flera pekare som pekar på samma minne men det kommer automatiskt att frigöra minne när alla pekarna går ur scopet. Dvs när ingenting längre pekar på minnet. På så sätt slipper man många av beskymrena kring pekare.

Det finns även `std::unique_ptr` och `std::weak_ptr`. Unique pointers tillåter endast att en variabel pekar på det minna som den är ansvarig för, och den frigör automatiskt minnet när den går ur scope. Unique pointers syfte är ofta att markera att programmet endast bör ha en referens till en variabel som har blivit allokerad på heapen.

17. Skapa en funktion `create_unique_pointer` som skapar en unique pointer till en int och returnerar den. Visa hur ownership överförs från en unique pointer till en annan.

18. Skapa en funktion `create_shared_pointer` som skapar en shared pointer till en int och returnerar den. Visa hur du kan ha flera variabler som alla pekar på samma sak.

Asserts

Om ditt program gör något väldigt fel brukar det bara krascha. Men om det gör något lagom fel finns det en risk att det istället fortsätter köra, men gör inte som

det ska. Kommandot `assert` låter kolla för att något är sant, och om det är inte sant, så kraschar programmet. Exempel på rimliga användningsmönster:

```
void perform_op(int op_type)
{
    if (op_type == 0) do_0();
    else if (op_type == 1) do_1();
    else if (op_type == 2) do_2();
    else assert(false);
}
```

I en graf vill man ibland ha:

```
void add_edge(int a, int b)
{
    assert(a!=b);
    ...
}
```

I en funktion som hittar största värdet i ett intervall av en array:

```
void interval_max(int a, int b)
{
    assert(a<=b);
    ...
}
```

För vissa icketriviala lösningar kan man orsaka undefined behavior om den asserten inte håller. Då kan man alltså inte alls lita på att det ska krascha.

Ibland är det inte trivialt att avgöra om det gått fel i början av en funktion. Då kan man blanda in asserts under algoritmens körning, och om man märker att det går katastrofalt fel så avslutas allt.

Viktigt är däremot att assert endast ska användas som debugging-verktyg. Det ska exempelvis inte vara the last line of defense så att användares hemligheter inte läcks eller dylikt. Detta beror på att compilern kan välja att kompilera den till en no-op om man kör med -O2 eller högre. Detta betyder också att det är ytterst viktigt att inte skriva kod som har några sidoeffekter innuti en assert. Den koden kommer ju inte att köras med vissa kompileringsflaggor.

Lär dig lita på kompilatorn

C++ kompilatorn är riktigt bra på att optimera kod som du har skrivit. Det bästa du kan göra för att optimera din kod är att skriva en algoritm som är snabb. Men att optimera en rad i taget för att undvika vissa småsaker är ofta inte värt det. Ett vanligt exempel är att folk väljer att skriva `(x >> 1)` istället för `(x / 2)` då de tänker att det är snabbare. Saningen är dock att det kompileras till exakt samma kod. Så det är smartare att behålla mer lättläst kod och låta kompilatorn göra jobbet åt dig.

Det finns många andra exempel som kompilatorn kan optimera till bättre kod.

Till exempel kan den unrolla korta for-loopar, och ersätta en komplicerad beräkning med en konstant.

Dock är det inte alltid så att kompilatorn lyckas att optimera kod, så ifall man verkligen vill veta vad kompilatorn klarar av så får man läsa på mer om det.

Ett sätt man kan se vad kompilatorn gör med ens kod är att kompilera ner koden till assembler. Det kan man exempelvis göra på `https://godbolt.org/`. Där kan man direkt se vilken del av koden som översätts till vad i assembler. Vi hinner inte att gå igenom exakt hur man ska tolka assembler under dagens lektion, men jobbar man mycket med C++ så är det värt att åtminstone kunna lite grann.

19. Gå in på godbolt.org och sa vad följande kod översätts till i assembler. Teste först utan att ha några särsklida kompileringsflaggor, och lägg sedan till flaggan -O2 som kommer se till att koden optimeras mer. Kan ni se någon skillnad i vad funktionerna kompileras till? Utan att förstå vad varje rad assembler gör, kan ni gissa varför det blir så stor skillnad?

```
int compiler_dislikes_this(int n)
{
    int ret = 0;
    for(int i = 0; i < n; i++){
        ret += i;
    }
    return ret;
}

int compiler_likes_this()
{
    int n = 1000;
    int ret = 0;
    for(int i = 0; i < n; i++){
        ret += i;
    }
    return ret;
}
```

Structs/Klasser

Precis som andra språk så finns det klasser i C++. Men speciellt för C++ är att det finns två olika val: `struct` och `class`. Den enda skillnaden i vad de egentligen betyder är att en structs innehåll är public per default vilket betyder att man får accessa alla variabler som inte explicit markeras som private eller protected. I en class är default att variabler är private istället.

Dock brukar structer och klasser användas på olika sätt. T.ex. enligt googles styleguide så används bara structs som en container för att spara data av olika typer och gruppera ihop dem. Klasser används för mer komplicerade saker, och där kan man lägga in metoder, arv och andra en massor av andra kul saker man

kan tänka sig göra.

Här är ett exempel på hur en simpel klass kan se ut:

```
class Person {
private:
    // These variables are private. They can only be accessed from the methods
        inside the class. This prevents users from modifying things that they
        should not be modifying.
    std::string name;
    int age;

public:
    // The functions down here are public, meaning they can be called from
        other places in the code.

    // Constructor
    Person(const std::string &name, int age) : name(name), age(age)
    {
        std::cout << "Constructor called: " << name << " is created.\n";
    }

    // Destructor. This is called when the variable goes out of scope, or when
        you call delete.
    ~Person() {
        std::cout << "Destructor called: " << name << " is destroyed.\n";
    }

    // Method to display person's information.
    void displayInfo()
    {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};
```

20. Skriv ett klass för att representera en punkt i 2d. Skapa en vector som innehåller intanser av din klass, och sortera sedan denna vektor i stigande ordning, först baserat på x-koordinat, och sedan på y-koordinat ifall de har samma x-koordinat.

21. Tidigare gick vi igenom vector, och hur pekare/smarta pekare fungerar. Välj antingen `vector` eller `shared_ptr` och implementera den klassen från scratch, för en valfri typ (t.ex. bara för int). Eran version bör inte ha stöd för alla metoder som finns i `std::vector` eller `std::shared_ptr`, men ni bör se till att hålla koll på minnet internt.

Templates

Låt säga att du skriver ett program där du använder arrayer av tal och du behöver ofta garantera att alla är positiva. Då kanske du skriver:

```
void make_positive(vector<int> &numbers)
```

```
{
    for (size_t i = 0; i < numbers.size(); i++)
    {
        numbers[i] = abs(numbers[i]);
    }
}
```

Allt är frid och fröjd. Men en dag behöver du använda den med flyttal:

```
void make_positive(vector<double> &numbers)
{
    for (size_t i = 0; i < numbers.size(); i++)
    {
        numbers[i] = abs(numbers[i]);
    }
}
```

Men oj, nu ber chefen dig att ibland använda normala flyttal för att spara på minne. Blir det ännu en overload? Nej, nu får det vara nog. Vi kan nämligen skriva en version som funkar för alla datatyper:

```
template<typename T>
void make_positive(vector<T> &numbers)
{
    for (size_t i = 0; i < numbers.size(); i++)
    {
        numbers[i] = abs(numbers[i]);
    }
}
```

Denna tar alltså en implicit parameter T, vilket den listar ut från vektorn vi anropar funktionen med. T kan användas precis som någon annan typ, såsom int eller annat innanför funktionen.

Exempel 2: man kan även template:a klasser, vilket är hur i princip hela C++ standardbiblioteket funkar.

```
template<typename T>
struct Node
{
    T value;
    Node *next;
};
```

Vilket vi använder genom

```
Node<int> *root = new Node<int>();
```

22. Skriv en egen linked list-implementation som använder templates. Denna måste endast stödja att lägga till saker på slutet och få värdet av det i:te elementet.

Exempel 3: här kommer ett riktigt exempel som Joshua använt. Låt säga att du vill använda gcc:s datastruktur order statistics tree. Detta är en map som bland annat låter dig snabbt svara på ”hur många nycklar har värde $\leq x$” i $O(log_2(N))$, där N är antalet nycklar. Om man använder vanliga std::map tar detta $O(N)$. Låt oss definiera denna, där vi har int som nyckel och värde.

```
#include <bits/extc++.h>
using namespace __gnu_pbds;
typedef tree<int, int, std::less<int>, rb_tree_tag,
    tree_order_statistics_node_update> indexed_map;
```

Här ser vi redan ett trick: för att slippa skriva hela det böset varje gång vi vill skapa en variabel av den typen kan vi använda typedef. Varje gång vi vill använda den räcker det nu med:

```
indexed_map map_example;
```

Men åh nej! Nu vill vi ändra våra nycklar till double. Ska vi copy-pastea den och skapa en ny version? Nej, vi kan templatea typedef:en!

```
template<typename T, typename U>
using indexed_map = tree<T, U, std::less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
```

Och nu kan vi bara skriva

```
indexed_map<double,int> double_map;
```

Interagera med C kod

Ibland händer det att man har lite kod som är skriven i C och annan kod som är skriven i C++. Som tur är så är det lätt att anropa kod som ligger i en fil skriven i det andra språket. Det räcker egentligen med att man inkluderar header filen som deklarerar funktionerna man vill använda, men man måste dock se till att använda göra det på följande sätt:

```
extern "C" {
#include "my_cool_header_file.h"
}
```

Anledningen till att man måste skriva `extern "C"` är att C++ använder sig utav nånting som kallas för name mangling. Utan att gå in på för mycket detaljer så innebär det att kompilatorn inte lyckas lista ut vilken funktion den borde anropa. Att skriva `extern "C"` löser dock det problemet.

23. Testa att skriva en fil i C som innehåller en funktion som kvadrerar ett heltal, och en fil i C++ som anropar denna funktionen.

24. Testa att skapa en C-funktion som beräknar längden av en sträng. Skapa en `std::string` i C++ och anropa funktionen som du skapade i C. Hur kan du göra

det, när `std::string` inte finns i C?

Compile-time template trickery

Rutinerade tävlingsprogrammerare har sedan länge tröttnat på skriva följande för att läsa input:

```
int a, b, c;
cin >> a >> b >> c;
```

Egentligen är det enda vi behöver specificera är typen och namnen på variablerna. En template:ad funktion kan (som tur är) inte skapa variabler där vi anropar den ifrån. Vi kan istället använda C++ preprocessor. Om vi exempelvis skriver

```
#define read(type, name) type name; cin >> name
```

Kan vi sedan skriva

```
read(int, a);
```

Notera att den inte tar typer på argumenten. Detta är för att preprocessorn inte vet att C++ finns: den gör i princip bara blind copy-pasting.

Men tänk om vi vill läsa in 2? 3? 4? Vi kan göra en version för varje antal med method overloading. Eller så kan vi göra häxkonst!

```
void readinput() {} // Recursion base case
template<typename T, typename... Args> void readinput(T& arg, Args&... args)
{
    cin >> arg;
    readinput(args...);
}
#define read(type, ...) type __VA_ARGS__; readinput(__VA_ARGS__);
```

__VA_ARGS__ innebär att vi tar godtyckligt många argument. Sedan anropar vi readinput med de.

Readinputs template tar som input en variable med godtycklig typ, och sen en lista med variabler som är samlade under Args. Då läser den först in variabeln och anropar sig själv med resten av argumenten. När den anropar sig själv extraheras ännu en gång den första variabeln ut ur listan, och den krymper med 1. Detta fortsätter tills den är tom och når base caset i toppen.

25. Försök förstå koden till fullo.

26. Implementera en print-funktion likt pythons, där du kan printa godtyckligt många argument av olika typer, mellanslagsseparerat. Använd bara templates.

27. Använd templates för att beräkna fakulteter vid kompileringstid.

28. Låt säga att du vill skapa en 4d-vektor av ints, där varje dimension har storlek a,b,c,d. Då måste du orioniskt skriva `vector<vector<vector<vector<int>>>> arr = \`
`vector<vector<vector<vector<int>>>>(a, vector<vector<vector<int>>>(b, vector<vector<int>>(c,`

`vector<int>(d))));`

Förenkla detta med en template! Den ska ta typen av vectorn, och skapar den en till dimension för varje variabel, med motsvarande storlek. Så exempelvis ska ovanstående kunna ersättas med

`multi_init(int, a, b, c, d);`

Tips: du kan ha return type auto. std::decltype är väldigt användbar.

Operators

Ifall man har en egen klass, och man vill att den ska stöjda saker som till exempel addition, bitshifting, eller indexering, så tillåter C++ att man skriver vad som ska hända i såna fall. Detta gör man genom att definera en funktion precis som vanligt, men syntaxen är aningen annorlunda.

```
class Point {
private:
    int x, y;

    Point(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

    Point operator+(Point other_point)
    {
        return Point(x + other_point.x, y + other_point.y);
    }
};
```

Man skriver alltså först typen som operatorn ska returnera följt av ordet operator, och sist den symbol som skrivs när man vill använda sig utav den funktionen. Ett par alternativ som finns är `*`, `+`, `-`, `/`, `>>`, `<<`, `[]`, `()`, `==`, `!=`, `%`, `+=`.

29. Skriv en klass för att representera bråktal. Erat program bör stödja åtminstone de fyra enkla räknesätten.

Cool overloading

Låt säga att du vill ha 2D-vektorer, men är för lat för att skriva en egen klass. std::pair<int,int> är nästan perfekt, men du vill kunna addera de. Då kan man lätt implementera detta, trots vi inte äger std::pair!

```
template <typename T, typename U> inline pair<T, U> operator+(const pair<T, U>
    l, const pair<T, U> r)
{
    return { l.first + r.first, l.second + r.second };
}
```

Tack vare att vi använder templates har vi nu definierat + för alla möjliga

pairs.

Undefined behavior

I c++ finns det många skumma saker som kan hända. Skriver man exempelvis följande kod, så kommer man med vissa kompilatorflaggor att lyckas köra programmet:

```
int arr[5];
cout << arr[20] << endl;
```

Programmet skulle kunna krascha då vi går out of bounds i vår array, men det skulle också kunna vara så att vi skriver ut det som finns där i minnet. Läskigt!

Har man kod som gör nånting som i detta exempel så innebär det förmodligen att man har en bugg i sitt program eller att man har kodat nånting på ett dåligt sätt. Så oavsett, så bör man undvika detta.

Men, det som gör sånt här extra intressant i C++ är att det finns någonting som heter undefined behavior. Det är egentligen en samling med saker som är förbjudna att göra enligt språket, och ifall man gör det så få kompilatorn dessutom göra vad den vill med den koden. Exempelvis så är overflow på signed integers undefined behavior. T.ex.:

```
int x = 1000000;
x = x * x; // Invokes undefined behavior because of overflow.
```

Ofta får man höra att overflow är en sak som finns i C++ och att det gör så att en variabel *wrappar runt* - Tobias Glimmerfors. Men det är även så att det är undefined behavior. Kompilatorn kan alltså göra något annat med koden, och enligt standarden så är erat program ett ogiltigt C++ program ifall ni har undefined behavior. Så akta er för det.

30. Vad gör följande kod? Försök först att tänka ut vad som kommer hända. Testa sedan att kompilera/köra den här koden med olika optimeringsnivåer. Testa -O0, -O1, -O2, och -O3.

```
int x = 2147483647 - 10; // 10 less than int max.
for(int i = 0; i < 20; i++){
    cout << x << endl;
    x++;
}
```

31. Förra problemet innehåller undefined behavior, så att nått skumt händer är faktiskt tillåtet. Skriver man sån kod så får man helt enkelt skylla sig själv då man skrev nånting som inte följer språkets standard. Men varför blir det så fel i förra uppfigten? Exeprimentera lite med koden i godbolt.org och se ifall ni kan lista ut vad kompilatorn försöker att göra.

Det finns många andra saker som är undefined behavior. T.ex. är det otillåtet att inkrementera en pekare så att den pekar utanför dett minne som den skapades

för. Det är också otillåtet att casta pekare till andra typer och sedan avreferera de....

32. Läs på mer om saker som är undefined behavior online. Försök att hitta nånting obskyrt som Tobias inte känner till, och lär honom något :)

Mer om macron

Macros är egentligen bara copy paste av kod, och detta görs som del av ett så kallat preprocessing steg. Dvs innan koden faktiskt kompileras så händer detta.

Ett vanligt exempel som tävlingsprogrammerar gillar är:

```
#define rep(i,a,b) for(int i = (a); i < (b); i++)
```

Då kan man skriva `rep(i, 0, n)` för att loopa från 0 till och med n - 1, vilket förkortar mängden man behöver skriva.

Det finns en del viktiga saker man bör veta om macron ifall man vill skriva ett macro.

33. Vad skriver följande kod ut? Varför? Tänk igenom ditt svar först innan du testar.

```
#define push_back_twice(vec, x) vec.push_back(x); vec.push_back(x);
int main()
{
    vector<int> vec;
    int x = 5;
    push_back_twice(vec, x++);
    cout << vec[0] << ", " << vec[1] << endl;
}
```

34. Vad gör följande kod? Försök först att lista ut det själv.

```
#define is_equal(a, b) a == b

int main()
{
    cout << (is_equal(5, 5)) << endl;
    cout << (is_equal(0 || 2, 2)) << endl;
}
```

Som ni ser så måste man vara försiktig när man skriver macros. Ifall man skickar in ett argument till ett macro som har en sidoeffekt så kommer den sidoeffekten att upprepas varje gång det används innuti macrot, vilket man förmodligen inte vill. Detta kan undvikas genom att inte skicka in saker med sidoeffekter till macron.

Man bör också se till att lägga in paranteser runt variablerna man använder innuti macron, om man inte är säker på att det verkligen inte behövs. Alltså borde man egentligen skriva:

```
#define is_equal(a, b) (a) == (b)
```

På så sätt kommer förra exempelt att fungera mer så som man tänker sig att det borde.

Macros kan även skrivas lite som funktioner, där de exekverar flera påståenden. Detta behöver inte användas så mycket i C++, men är väldigt användbart i C ibland. T.ex.:

```
#define calc_dist(x, y, x2, y2) ({\
    typeof(x) dx = (x) - (x2);\
    typeof(x) dy = (y) - (y2);\
    sqrt(dx * dx + dy * dy);
})
```

Vill man skriva ett macro på flera rader så behöver man lägga in ett backslash i slutet av varje rad förutom den sista. I exemplet `calc_dist` så är det det på den sista raden som blir output när man använder macrot.

Ifall man vill skriva ett macro som inte returnerar nånting så brukar man skriva det på följande sätt:

```
#define some_cool_macro_name(vec, x) do {\
    /* Put some interesting code here */\
} while(0)
```

Det som är innuti do-while kroppen kommer bara att köras en gång eftersom det står `while(0)`, så då blir det som att köra en vanlig funktion. Att det inte finns något semikolon i slutet av macrot gör att man själv måste skriva det när man använder macrot, vilket leder till att macrot känns som att använda en funktion.

Det finns mycket mer man kan lära sig om macron, så det är bara att fråga ifall ni är intresserade.

Sammanfattning

Det finns massor av saker man kan lära sig om C++. Här har vi nu försökt att gå igenom massor av olika saker ytligt. Vill ni lära er mer om C++ rekommenderar vi att ni läser på mer om de features som ni tyckte verkade mest intressanta.