תכנות מתקדם

7 בפברואר 2020

מרצה: ד"ר פנחס וויסברג

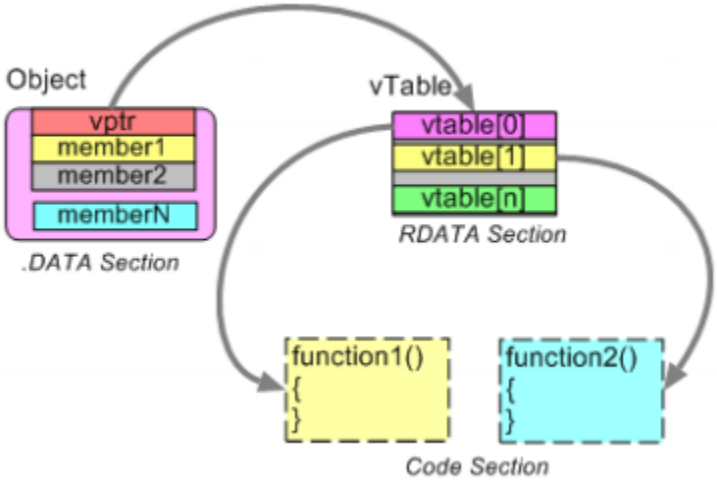
סיכום זה מבוסס על המצגות והקלטות השיעורים , הסיכום על אחריות המשתמש.

תודה גדולה לאביהוא אושרי על הצילומים

שימו לב: בסיכום יש לא מעט צילומי מצגות **בהדפסה** הם עלולים לצאת לא כל כך ברור, לשיקולכם.

בהצלחה!!

לתיקונים/הערות - נעם דומוביץ 0508752542



תוכן עניינים

הערה: חסר שיעור 9 - פנחס לא הגיע.

שיעור 1

1 מבוא - רענון ותוספות

1.1 תוספות של $c++11$

1.1.1 $auto$

auto אומר לקומפיילר להסיק את סוג המשתנה לפי המאתחל:

```
auto b = true;        // bool
auto ch = 'x';        // char
auto i = 123;         // int
auto d = 1.2;         // double
auto p = &i;          // pointer to int
auto& ri = x;         // reference on x
const auto& cri = x;  // constant reference on x
auto z = sqrt(y);     // whatever sqrt(y) returns
auto item = val1 + val2; // result of val1 + val2
```

מה ההבדל בין $refernce$ ל $pointer$?

- אם יש לנו $pointer$ ואנחנו רוצים לגשת למשתנה צריך לכתוב $*$, ו$refernce$ לא צריך (בדוגמה נוכל לגשת לri או ל x)
- ב $pointer$ ניתן בהמשך התכנית להגדיר שיצביע על משהו אחר, $refernce$ אי אפשר

$const\,refernce$ - מגדיר משתנה הניתן לקריאה ולא מאפשר כתיבה

נפח הזיכרון לכל סוג משתנה:

ייצוג נתונים

C Data Type	Typical 32-bit	Typical 64-bit
char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
pointer	4	8

- נשים לב להבדל בין הגודל של ה $pointer$ בין המערכות, היות וזה נובע ישירות מחלוקת הזיכרון

1.1.2 $decltype()$

הוספות של C++11 - decltype()

decltype() אומר לקומפיילר להסיק את סוג המשתנה מתוך ביטוי שאינו משמש לאתחול:

```
decltype(f()) sum = x;  // whatever f() returns
const int ci = 0;
decltype(ci) x = 0;     // const int
struct A { int i; double d; };
A* ap = new A();
decltype(ap->d) x;      // double
vector<int> ivec;
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix); // vector<int>::size_type
```

דומה ל $auto$, רק שאומר לקומפיילר להבין את סוג המשתנה מתוך ביטוי שנמצא במקום אחר

1.1.3 4 סוגי אתחול

אתחול vector:

```
vector<int> v{1,2,3,4,5,6,7,8,9};
vector<string> articles = {"a", "an", "the"};
vector<int> v1(10); // v1 has ten elements with value 0
vector<int> v2{10}; // v2 has one element with value 10
vector<int> v3(10, 1); // v3 has ten elements with value 1
vector<int> v4{10, 1}; // v4 has two elements, 10 and 1
```

ארבע אפשרויות לאתחול משתנה:

```
int units_sold = 0;
int units_sold = {0};
int units_sold{0};
int units_sold(0);
int i1 = 7.2; // i1 becomes 7
int i2{7.2};  // error : narrowing conversion
```

1.1.4 $range\ for$

```
vector<int> v{1,2,3,4,5,6,7,8,9};
for (auto i : v)          // i is a copy
     cout << i << " ";
cout << endl;
for (auto &i : v)       // i is a reference
    i *= i;             // square the element
```

- נשים לב להבדל בין ה for־ים , בראשון i הוא **העתק** של האובייקט, ובשני זה ה**ערך עצמו**

1.1.5 $enum$

```
enum class Color { red, blue , green };
enum class Traffic_light { green, yellow, red };
Color col = Color::red;
Traffic_light light = Traffic_light::red;
Color x = red;                  // error, which red?
Color y = Traffic_light::red; // error, not a Color
Color z = Color::red;           // OK
int i = Color::red;             // error, not an int
Color c = 2;                    // error, 2 is not a Color
// enum with no class are not scoped within their enum
enum color { red, green, blue };      // no class
enum stoplight {red, yellow, green}; // error, redefines
int col = green; // convert to their integer value
```

enum

- כאשר מגדירים $enum$ שונים, מומלץ לתת שמות שונים, ולכן ב $c++11$ יש את האפשרות ל $enum\ class$, וניתן (ואפילו הכרחי) לגשת במפורש לשמות דומים מ$class$ שונים.

1.2 $string$

string הוא מחרוזת של תווים בגודל משתנה.

הגדרה ואתחול של **string**:

```
string s1; // default initialization, empty string
string s2 = s1;      // copy of s1
string s2(s1);       // copy of s1
string s3 = "abc";   // copy of the string literal
string s3("abc");    // copy of the string literal
string s4(10, 'c'); // cccccccccc
```

פעולות על **string**:

```
s.size()             // number of characters in s
s[n]                 // reference to char at position n
s1 + s2              // concatenation of s1 and s2
```

- הרעיון הוא שלקחו את ה $cstring$ כלומר ה $string$ מ c
- החסרון שהגדול קבוע ואי אפשר להגדיל/להקטין ושאר פעולות
- לכן כעת יש אובייקט $string$, ואותו כבר ניתן להגדיל , והוסיפו לא כל מיני פעולות כמו האופרטור $+$
- בנוסף יש מספר אפשרויות אתחול - נובע ממספר ה $construcr$ שהגדירו ל $string$

1.2.1 $size\_type$

```
string::size_type len = line.size();
// size() returns a string::size_type value

The string class defines size_type so we can use the
library in machine independent manner

We use the scope operator to say that the name
size_type is defined in the string class

It is an unsigned type big enough to hold the size of
any string
```

- בתוך המחלקות $string$ ו $vector$, הגדירו משתנים , וניתן להגדיר אותם מטיפוס מסוים, וכך עשו עבור גודל ה$string$, וכפי שמופיע במצגת זו הדרך הנכונה להצהיר על משתנה שיחזיק אורך/גודל של $string$

דוגמאות למעבר על $string$:

- לפי אינדקס:

```
string s("some string");
// The subscript operator (the [ ] operator)
// takes a string::size_type
for (string::size_type index = 0;
     index != s.size() && !isspace(s[index]);
     ++index)
          s[index] = toupper(s[index]);

The output of this program is:
SOME string
```

- עם $range$

```
// convert s to uppercase
string s("Hello World!!!");
for (auto &c : s)      // note: c is a reference
     c = toupper(c);   // so the assignment changes
cout << s << endl;

The output of this program is:
HELLO WORLD!!!
```

- קליטת מחרוזות מהמשתמש :

קריאת מספר לא ידוע של מחרוזות

```
string word;
// end-of-file or invalid input will put cin in error state
// error state is converted to boolean false
while (cin >> word)
     cout << word << endl;
```

קריאת שורה שלמה:

```
string line;
// read up to and including the first newline
// store what it read not including the newline
while (getline(cin, line))
     cout << line << endl;
```

דוגמה לבעיה:

```
int x;
while (cin >> x )
    cout << x;
```

עבור קלט: 123 456 יפול ב ABC

ABC

הסבר: הcin מחזיק בנוסף משתנה של $true\backslash false$ וכאשר מקבלים קלט לא תקין או סוף קובץ הוא הוא הופך ל$false$

שיעור 2

1.2.2 $vector$

בשיעור שעבר הצגנו את האובייקט $String$ שהוא נותן לנו הרבה יותר אפשרויות מאשר $Cstring$ שזה היה מערך של $char$ים בC , כך גם $vector$ הוא למעשה מערך, אבל נותן לנו הרבה יותר אפשרויות :

vector הוא אוסף של אובייקטים מאותו סוג בגודל משתנה.

הגדרה ואתחול של vector:

```
A vector is a class template
We have to specify which objects the vector will hold
vector<int> ivec;              // initially empty
vector<Sales_item> Sales_vec;
vector<vector<string>> file;  // vector of vectors
```

פעולות על vector:

```
v.size()        // number of elements in v
v[n]            // reference to element at position n
v.push_back(t)  // add element to end of v
```

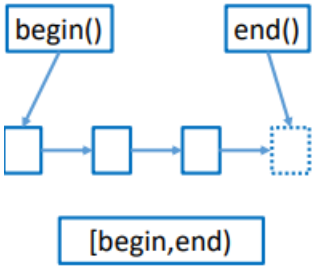
הוספת אלמנטים למיכל vector:

```
vector<int> ivec;          // empty vector
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
     ivec.push_back(ix); // adds element with value ix
```

טעויות נפוצות:

```
vector<int> ivec;
cout << ivec[0];        // error: no elements
vector<int> ivec2(10);
cout << ivec2[10];      // error: elements 0 to 9
vector::size_type       // error
vector<int>::size_type // ok
```

1.2.3 איטרטורים



איטרטור הוא אובייקט שמצביע על איבר של הסדרה
האיטרטור begin מצביע על האיבר הראשון
האיטרטור end מצביע על המקום שאחרי האיבר האחרון
איטרטור צריך לספק את הפעולות הבאות:
השוואה בין שני איטרטורים, האם מצביעים לאותו איבר:
`iter1 == iter2 , iter1 != iter2`

התיחסות לערך של האיבר שהאיטרטור מצביע עליו:
`val = *iter , *iter = val`
`(*iter).member` במקום `iter->member`

קידום האיטרטור כך שיצביע לאיבר הבא:
`++iter`

- ישנם איטרטורים שמספקים פעולות נוספות

דוגמה עם : *string*

```
// change the case of the first word in a string
string s("some string");
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)

    *it = toupper(*it);

The output of this program is: SOME string
```

1.2.4 סוגי איטרטורים

לכל המיכלים יש איטרטורים , הם מגודרים בספריה שכל מיכל

```
vector<int>::iterator it; // it can read and write
string::iterator it2; // it2 can read and write
vector<int>::const_iterator it3; // can read but not write
```

```
string::const_iterator it4; // can read but not write
// The type returned by begin and end depends on whether the object is const
auto it1 = v.begin(); // It is best to use a const when we need to read only To ask for the const_iter
type auto it3 = v.cbegin(); // it3 has type const_iterator
```

דוגמה - חיפוש בינארי באמצעות איטרטורים

```
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg) / 2;
while (mid != end && *mid != sought) {
    if (sought < *mid) end = mid; // if so, adjust the range
    else // element is in the second half
        beg = mid + 1; // start looking just after mid
    mid = beg + (end - beg) / 2; }
}
```

סוף שיעור 2 - תחילת 3

1.3 מיכלים

- $string$ מיכל שמכיל תווים בלבד. גישה אקראית מהירה, הוספה ומחיקה מהירה בסוף המחרוזת.
- $vector$ מערך בגודל משתנה. גישה אקראית מהירה, הוספה ומחיקה מהירה בסוף הוקטור.
- $deque$ תור עם שני קצוות. גישה אקראית מהירה, הוספה ומחיקה מהירה בתחילת ובסוף התור.
- $list$ רשימה מקושרת כפולה. גישה סדרתית בלבד, הוספה ומחיקה מהירה בכול מקום ברשימה. (חסרון א"א לגשת בצורה אקראית)
- map אוסף של זוגות מפתח־ערך. חיפוש מהיר של ערך לפי המפתח. (מימוש בcpp : עץ חיפוש מאוזן, או $hashtable$)
- set אוסף של מפתחות. חיפוש מהיר האם מפתח נמצא באוסף

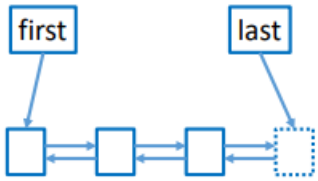
תזכורות:

- היתרון בהגדרת אובייקטים בסגנון $string$ או $vector$ היא הקצאת זיכרון דינאמי, ואפשרות להוספת מתודות

1.3.1 אוסף פונקציות

ל$vector$ אין $push\_front$, אבל ל:

push_front



למיכל **list** ולמיכל **deque** אפשר להוסיף במהירות איבר גם לתחילת הרשימה עם **push_front**:

```
list<int> ilist;
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

הכנסה בכל מקום, ניתן להכניס גם כמה דברים,

insert

הוספת איבר או איברים בכל מקום במיכל.

הפרמטר הראשון הוא איטרטור שמצביע על המקום **שלפניו** יוכנסו האיברים:

```
list<string> slist;
// equivalent to calling slist.push_front("Hello!")
slist.insert(slist.begin(), "Hello!");
// insert the last two elements of v at the beginning of slist
slist.insert(slist.begin(), v.end() - 2, v.end());

vector<string> svec;
// no push_front on vector but can insert before begin()
svec.insert(svec.begin(), "Hello!");
// Can insert anywhere in a vector or string
// However, doing so can be an expensive operation
```

יכניס לפני האיטרטור

נשים לב ל$overloading$ של $insert$

emplace

בנית אלמנט באמצעות בנאי והכנסתו למיכל.

```
// construct a Sales_data object at the end of c
// uses the three-argument Sales_data constructor
c.emplace_back("978-0590353403", 25, 15.99);
// equivalent to creating a temporary Sales_data object
// and passing it to push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));

c.emplace(iter, "999-999999999", 25, 15.99);
// The arguments to emplace must match a constructor
```

map

ספר טלפונים :

```
map<string,int> phone_book {
   {"David Hume",123456},
   {"Karl Popper",234567},
   {"Ber trand Arthur William Russell",345678}
};
int get_number(const string& s)
   { return phone_book[s]; }
// When indexed by a key, a map returns the value
// If a key isn't found, it is entered into the map with a
// default value
// To avoid entering invalid numbers into our phone book,
// we could use find() instead of []:
phone_book.find(s)
```

$find$ מחזיר איטרטור לזוג (key, val)

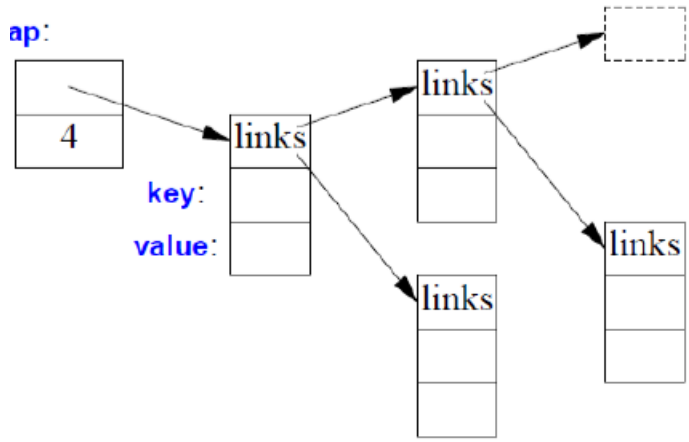
שיעור 3

דוגמה לשימוש בmap לספירת הופעות מילים בקלט:

```
map<string, size_t > word_count ; // empty map
string word;
while ( cin >> word)
    ++ word_count [word];
for const auto &w : word_count ) // for each element in map
    cout << w.first << " occurs " << w.second << w.second > 1 ) ? " times" : " time") << endl
The output of this program is:
    Although occurs 1 time
    Before occurs 1 time
    // Elements in a map are of type pair A pair holds two data members first and seconde
```

שימה אינו יעיל, מחיר החיפוש הוא O(n)

עץ חיפוש מאוזן, ומחיר החיפוש הוא O(log(n))



SET

דוגמה: מספר פעמים שמילה מופיע ללא מילים שכיחות

```
map<string, size_t > word_count;
set<string> exclude = { the ", "and ", "or", "an ", a"};
string word;
while ( cin >> word)
// count only words that are not in exclude
    if ( exclude.find (word) == exclude.end
        ++ word_count [word];
```

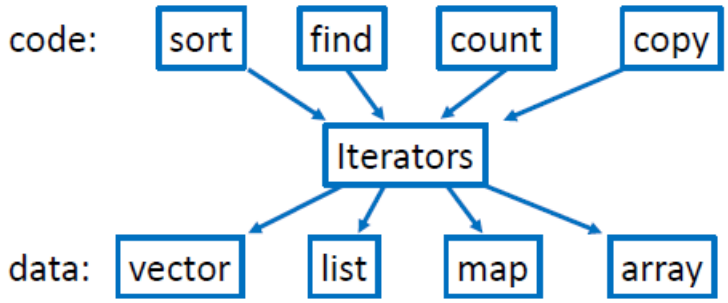
אלגוריתמים

המיכלים מגדירים מספר קטן של פעולות: הוספה, מחיקה, גודל.

ישנם פעולות נוספות שנרצה לעשות: חיפוש איבר, החלפת איבר, סידור מחדש של האיברים.

כדי שלא נצטרך להגדיר את כל הפעולות עבור כל המיכלים, הספרייה הסטנדרטית (STL) הגדירה אלגוריתמים שיכולים לפעול הרבה מיכלים.

המיכל מספק איטרטור, האלגוריתם קורא וכותב את הנתונים שבמיכל באמצעות האיטרטור.



יצרו מתודות שמתאימות לכל המיכלים, והמגשר זה איטרטורים

1.4.1 $find$

find

find מחפש האם המיכל מכיל ערך מסוים.

האלגוריתם עובר על המיכל בתחום של שני איטרטורים:

```
int val = 42; // value we'll look for
// result will denote the element we want if it's in vec,
// or vec.cend() if not
auto result = find(vec.cbegin(), vec.cend(), val);
cout << "The value " << val
<< (result == vec.cend()
? " is not present" : " is present") << endl;
```

find

האלגוריתם עובר על המיכל בתחום של שני **איטרטורים**,
לכן אפשר להשתמש בו גם כדי לחפש במיכלים אחרים:

```
// look through string elements in a list
string val = "a value";
auto result = find(lst.cbegin(), lst.cend(), val);
```

מצביעים פועלים כמו **איטרטורים**, לכן אפשר לחפש גם **במערך**:

```
int ia[] = {27, 210, 12, 47, 109};
int val = 83;
int* result = find(ia, &ia[5], val);
// int* result = find(begin(ia), end(ia), val);
```

אפשר לחפש **בחלק** מהמערך:

```
// search from ia[1] up to but not including ia[4]
auto result = find(ia + 1, ia + 4, val);
```

מה קורה כאשר עושים *include* ?

- שאלה נניח יש קובץ A וקובץ B, שקימפלת ביחד איך ניתן בקובץ A להשתמש בפונקציה f מקובץ B ?
- תשובה: צריך לרשום בראש הקובץ את חתימת הפונקציה

זה מציק. לכן יש את *header* כפי שאנחנו מכירים.

- אבל מה קורה כאשר רוצים להביא פונקציות מספריות גדולות? רצו לפטור את המתכנת מהצורך לרשום את כל חתימות הפונקציות, וזה מה שעושים ה *include* .
- אבל כאשר אנחנו עושים *include algorithm* , מאיפה יש לו את הפונקציה הרלוונטית?
- תשובה: כל הדברים הללו קשורים לספריה הסטנדרטית שנרחיב עליה בהמשך.

1.4.2 accumualte

accumulate

accumulate מסכם את האיברים שבמיכל.
עובר על המיכל בתחום של שני **איטרטורים**, ומוסיף אותם לפרמטר השלישי.
הפרמטר השלישי קובע איזו פעולת חיבור תתבצע:

```
// sum the elements in vec starting the summation with 0
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
```

הפעולה + מוגדרת עבור string לכן אפשר לחבר **מחרוזות**:

```
string sum = accumulate(v.cbegin(), v.cend(), string(""));
```

שגיאה, הפעולה + **לא** מוגדרת עבור char *:

```
// error: no + on const char*
string sum = accumulate(v.cbegin(), v.cend(), "");
```

באלגוריתמים כמו find ו- accumulate שרק **קוראים**, עדיף להשתמש ב- cbegin ו- cend

שוב נשים לב להבדל בין $cstring$ לאוביקט $string$ של cpp

1.4.3 equal

equal

equal בודק האם שני מיכלים הם שווים ומחזיר אמת או שקר.

האלגוריתם מקבל שני איטרטורים עבור המיכל הראשון ואחד עבור המיכל השני:

```
// roster2 should have at least as many elements as roster1
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

אפשר להשוות בין מיכלים שונים, ואפילו בין מיכלים שלהם אלמנטים שונים ובלבד שאפשר להשוות ביניהם באמצעות ==.

לדוגמה:

```
list<string> roster1 = {"a", "an", "the"};
vector<const char*> roster2 = {"a", "an", "the"};
```

האלגוריתם מניח שהמיכל השני גדול לפחות כמו הראשון.

מיכלים שונים לדוגמה $vector$ מול $list$, כלומר מסוג שונה (בגלל שמשווים דרך איטרטור) , כנ"ל לגבי הערכים המושווים יכולים להיות מסוגים שונים. באותו אופן ניתן להשוות בין כל שני אובייקטים שמגודר בעבורם השוואה

1.4.4 copy

copy

copy מעתיק איברים ממיכל למיכל.

האלגוריתם מקבל שני איטרטורים עבור המיכל הראשון ואחד עבור המיכל השני:

```
// use copy to copy one built-in array to another
int a1[] = {0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)]; // a2 has same size as a1
// ret points just past the last element copied into a2
auto ret = copy(begin(a1), end(a1), a2);
```

$sizeof(*a1)$ - נותן את הגודל של האיבר הראשון, וכך ניתן לחלק בגודל המערך , כדי לדעת את מספר האיברים:

$sizeof$ - מחזיר גודל בבתים (ולא בביט).

1.4.5 sort, unique

sort, unique

sort ממיין את המיכל לפי האופרטור <.

unique מסדר את המיכל כך שבתחילת המיכל לא יופיעו איברים כפולים.

האלגוריתמים מקבלים שני איטרטורים:

```
void elimDups(vector<string> &words) {
// sort words alphabetically so we can find the duplicates
     sort(words.begin(), words.end());
// unique reorders the input so that each word appears once
// in the front portion of the range
// and returns an iterator one past the unique range
     auto end_unique = unique(words.begin(), words.end());
// erase uses a vector operation to remove the non-unique
     words.erase(end_unique, words.end());
}
```

unique - מבטל כפילויות לוקטור חדש, ולכן חייבים לרוקן את הערכים ה"ישנים" על ידי *erase* .

הערך המוחזר הוא *iterator* - למקום אחרי הערכים הקיימים

1.4.6 $replace, replace\_copy$

replace, replace_copy

replace מחליף איבר מסוים באיבר אחר בתוך המיכל.

replace_copy מחליף איבר מסוים באיבר אחר ומעתיק למיכל אחר.

מסדר את המיכל כך שבתחילת המיכל לא יופיעו איברים כפולים.

```
// replace any element with the value 0 with 42
replace(ilst.begin(), ilst.end(), 0, 42);

// leave the original sequence unchanged
// a third iterator is a destination to write the sequence
replace_copy(ilst.cbegin(), ilst.cend(), ivec.begin(), 0, 42);
```

צריך לוודא שבמיכל האחר יש מספיק מקום.

איך בכל המקרים נוודא שהמיכל המיועד יהיה ריק? היינו רוצים שפשוט יהיה מנגנון שידע לצור את המכיל בהתאם לגודל הדרוש.

back_inserter

אפשר להשתמש ב- back_inserter כדי להעתיק למיכל שאין בו מספיק מקום או למיכל ריק.

back_inserter מוסיף איברים למיכל ומעתיק לתוכם.

```
vector<int> vec; // empty vector
auto back_it = back_inserter(vec);
// assigning through back_it adds elements to vec
*back_it = 42; // vec now has one element with value 42
```

בדוגמה הקודמת:

```
vector<int> vec; // empty vector
// use back_inserter to grow destination as needed
replace_copy(ilst.cbegin(), ilst.cend(),
     back_inserter(ivec), 0, 42);
```

ולכן רואים בדוגמה שהעברנו $back\_inserter(ivec)$ כאשר $ivec$ הוא $vector$ ריק.

כיצד הוא פועל? זהו אובייקט עם העמסה של שלושה אופרטורים , $*$, השמה, $++$, ואז יצא שבכל שלב יתבצע $push\_back$

Insert Iterators

back_inserter יוצר אוביקט שמתנהג כמו איטרטור ומשתמש ב- push_back

front_inserter יוצר איטרטור שמשתמש ב- push_front

inserter יוצר איטרטור שמשתמש ב- insert

הפעולות המוגדרות עבור Insert Iterators:

```
it = t
// calls c.push_back(t), c.push_front(t)
// or c.insert(t,p) where p is the iterartor given to insert

*it, ++it ,it++
// Each operator returns it, they do nothing
```

בכיתה הראנו דוגמאות ל$find$ ו $replace$

$transfrom$ - כמו "העתקה לינארית" למכיל אחר

1.5 העברת פונקציה לאלגוריתמים

העברת פונקציה לאלגוריתמים

הרבה אלגוריתמים משווים איברים, ולצורך זה משתמשים באופרטורים < או ==.

לפעמים נרצה להגדיר בעצמנו מה קטן ומה שווה.

לאלגוריתמים יש גרסאות שמקבלות פונקציה שמחליפה את < או ==.

דוגמה, מיון לפי גודל מילה:

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
return s1.size() < s2.size();
}
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
```

מיון לפי אורך של *string* (ולא לקס')

1.6 פונקציית למבדה

העברת ביטוי למבדה לאלגוריתמים

כשמעבירים פונקציה לאלגוריתמים, צריך להגדיר אותה בנפרד.

מלבד זאת, הפונקציה לא יכולה לקבל יותר מאחד או שני פרמטרים שהאלגוריתם מעביר לה.

ביטוי למדה הוא פונקציה ללא שם שאפשר להגדיר בתוך הקריאה לאלגוריתם, יש לו את הצורה הבאה:

```
[capture list] (parameter list) { function body }
```

דוגמה, הדפסת איברי המיכל עם for_each:

```
// print words, each one followed by a space
for_each(words.begin(), words.end(),
     [](const string &s){cout << s << " ";});
cout << endl;
```

פונקצית למבדה, מבנה:

מתחילה ב [] - פרמטר שעובר על ידי התוכניות לאגלוריתם (למשל כי הוא מתגלה רק בריצת התוכנית) , מיד אחרי () - פרמטר שעובר לפונקציה , מיד אחרי {} - גוף הפונקציה

ביטוי למבדה

Capture by Reference מאפשר לביטוי למדה לשנות ערכים מחוץ לביטוי.

דוגמה, חשב את סכום האיברים ושמור אותו במשתנה שהוגדר מחוץ ללמדה:

```
int sum = 0;
for_each(vec.begin(), vec.end(),
     [&sum] (int x) { sum += x; });
```

דוגמה, הדפסת איברי המיכל:

```
// Print words to os separated by c:
void print_words(vector<string> &words,
     ostream &os = cout, char c = ' ')
{
   for_each(words.begin(), words.end(),
      [&os, c](const string &s) { os << s << c; });
} // the only way to capture os is by reference
```

הלמבדה כותבת אל מחוץ לפונקציה, ולכן צריך רפרנס.

דוגמה להעמסה על אופרטור () - קריאה לפונקציה:

העברת אובייקט פונקציה לאלגוריתמים

מחלקה שמעמיסה את אופרטור הקריאה לפונקציה, מאפשרת להשתמש באובייקטים של אותה מחלקה כאילו הם פונקציה.

דוגמה, אובייקט פונקציה שמחזיר את הערך המוחלט של מספר :

```
struct absInt {
     int operator()(int val) const {
          return val < 0 ? -val : val;
     }
};

int i = -42;
absInt absObj; // object that has a function-call operator
int ui = absObj(i); // passes i to absObj.operator()
```

בדוגמה קיבלנו אובייקט שמחזיר ערך מוחלט

2 בניית מחלקה

2.1 מחלקה

- מחלקה היא הרחבה של *struct* שבשפת C
- במחלקה ניתן להגדיר בנוסף למשתנים של *struct* גם פונקציות חברות במחלקה (*Functions Member*)
- מחלקה מאפשרת:
  - הפשטת נתונים (*Data Abstraction*) התעלמות מפרטי המימוש של העצם והתרכזות במאפיינים שלו
- כימוס (*Encapsulation*) הסתרת פרטי המימוש מהמשתמש
  - ניתן לקבוע הרשאות גישה לחברי המחלקה:
  - חברי מחלקה המוגדרים *private* נגישים רק לפונקציות חברות במחלקה • חברי מחלקה המוגדרים *public* נגישים גם לשאר פונקציות התכנית

דוגמה למימוש *vector* - נשים לב לאתחול על ידי *list initialization*

```
class Vector {
    int sz; // the size
    double* elem; // a pointer to the elements
public:
    using size_type = unsigned long;
    Vector(): sz{0},elem{nullptr} {} //default constructor
    Vector(int s) // constructor (s is the element count)
        :sz{s}, elem{new double[s]} // initialize
        { for (int i = 0; i<sz; ++i) elem[i] = 0.0; }
    ~Vector() // destructor
        { delete[] elem; }
    int size() { return sz; }
}; // end of class (struct)
    Vector v1; // use default constructor, not Vector v1();
    Vector v2(10); // create a vector with 10 elements
```

2.1.1 nullpter

- We try to ensure that a pointer always points to an object, so that dereferencing it is valid
- When we don't have an object to point to, we give the pointer the value nullptr
- In older code, 0 or NULL is typically used, but . . .

```
void func(int n); void func(char *s); func( NULL );
                  // which function is called? (int)
```

- using nullptr eliminates confusion between integers and pointers

```
func( nullptr ); // func(char *s) is called

double* pd = nullptr;
int x = nullptr; // error : nullptr is a pointer
```

פתרון למקרים בהם אנחנו רוצים לתת ערך לפויטר ללא ערך (ולא להשאר עם ערך רנדומלי)

פתרון לבלבול בין פוינטר לint

2.1.2 default constructor

- If our class does not explicitly define any constructors, the compiler will implicitly define the default constructor for us
- It default-initializes the members
- Objects of builtin or compound type (such as arrays and pointers) that are defined inside a block have undefined value

הקומפיילר מגדיר לבד את ה $defualt$ בתנאי שלא הגדרנו בנאי אחד, אם הגדרנו אחר ורוצים גם את הרגיל , אז:

- we can ask the compiler to generate the default constructor for us by writing = default

```
class Vector { Vector() = default ; };
```

- We are defining this constructor only because we want to provide other constructors

2.1.3 Conversion constructor

A constructor that takes a single argument defines a conversion from its argument type to its class

נסביר דרך דוגמה:

```
class complex {
    complex( double,double);
    complex(double);
    ...
};
complex z = complex{ 1.2,3.4  };
z = 5.6 ; // OK, converts 5.6 to complex( 5.6,0  and assigns to z
```

2.1.4 explict

הבעיה היא שההמרות הללו יכולות לצור תוצאות בלתי צפויות, ולכן ניתן למנוע זאת על ידי:

```
class Vector {
    explicit V ector( int
};
Vector v( 10 ); // OK, explicit
v = 40 ;    // error , no int to vector conversion
```

2.1.5 אתחול וקטור

- Initialize to default and then **assign**:

```
Vector v1(2); // error prone
v1[0] = 1.2 ; v1[1] = 2.4 ; v1[2] = 7.8
```

- Use **push_back**

```
Vector v2 ;// tedious
v2.push_back(1.2); v2.push_back (2.4 ); v2.push_back(7.8);
```

  - **push_back** is useful for input:

```
Vector read (istream & is) {
   Vector v;
   for (double d; is >> d;)
     v.push_back (d);
   return v
}
```

- Best **use { }** delimited list of elements:

```
Vector v3 = {1.2 , 7.89 , 12.34 };
```

מדוע הזכרנו זאת, כי נרצה שאפשרות כזה תהיה גם אצלנו ב*class* , ולכן צריך להגדיר *constractor* מתאים:

```
class Vector {
    int sz ; // the size
    double * elem ; // a pointer to the elements
public:
    Vector( initializer_list <double > lst ) // constructor
    :  sz lst.size ()}, elem {new sz ]} {
       copy ( lst.begin lst.end elem ); // copy using standard library algorithm
    }
};
```

2.1.6 Copy constructor

- A constructor is the copy constructor if its first parameter **is a reference** to the class:

```
Vector( const Vector& rhs ) ; // copy constructor
```

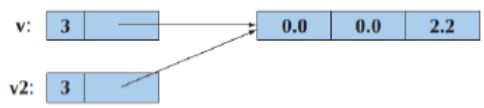
- copy constructor is used direct initialization and copy initialization

```
string s(dots); // direct initialization
string null_book = "99999"; // copy initialization
string nines = string( 100 ,'9'); // copy initialization
```

- Copy initialization happens also when passing an object to a function or returning an object from a function
- if we use an initializer that requires conversion by an explicit constructor:

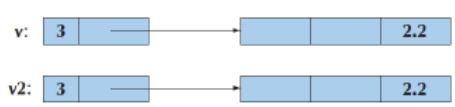
- באופן דיפולטיבי לאובייקט יש בנאי העתקה, אבל זו $member-wise-copy$ שהיא העתקה רדודה $(Shallow\ copy)$, וללא הגדרת בנאי ההעתקה זה מה שיתרחש:

```
Vector v2 = v
```



- לכן צריך להגדיר בנאי שיעשה $deep\,copy$ לכל האלמנטים של הוקטור, (נמשיך את הדוגמה של מחלקת $Vector$) שלנו. בנאי העתקה שמעתיק כראוי:

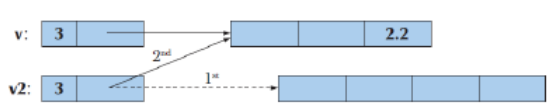
```
Vector( const V ector& rhs ) ; // copy constructor
    : sz {rhs.sz }, elem {new double[rhs.sz]}; {
        copy( rhs.elem , rhs.elem+sz , elem );
}
```



2.2 השמת העתקה

הבעיה:

```
Vector v(3);
v.set(2,2.2);
Vector v2(4);
v2 = v;
```

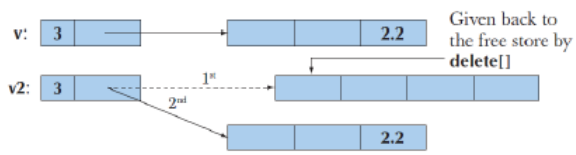


כלומר באופן דיפולטיבי פעולת ההשמה (אופרטור =) , יבצע ה $member-wise-copy$, שניהם **יצביעו** על אותו וקטור , ולכן:

- במחיקה יהיה $double\ deletion$
- יהיה $memory\ leak$ על הוקטור המקורי של $v2$

לכן, נוסיף ל$class$:

```
class Vector  {
    Vector & operator=(const Vector &) ; // copy assignment
    // . . .
};
Vector &Vector::operator=(const Vector & rhs) {
    double * p = new double[rhs.sz ]; // allocate new space
    copy( rhs.elem , rhs.elem+rhs.sz, p); // copy elements
    delete [] elem ; // deallocate old space
    elem = p; // now we can reset elem
    sz = rhs.sz
    return *this; // return a self reference To be consistent with built in types
}
```



הערה: אם פועלים על פי סדר פעולות זה, לא נצורת בעיה ב $self-assinment$ $(v = v;)$

נזכיר את ההגדרות:

- **Shallow copy** copies only a pointer so that the two pointers now refer to the same object
- **Deep copy** copies what a pointer points to so that the two pointers now refer to distinct objects

2.2.1 מניעת העתקה

למשל נשתמש כאשר נבנה $singleton$, דוגמה:

```
struct NoCopy {
    NoCopy () = default ; // use the synthesized default constructor
    NoCopy const NoCopy &) = delete ; // no copy
    NoCopy &operator=(const NoCopy &) = delete ; // no assignment
    ~NoCopy () = default ; // use the synthesized destructor other members
};
```

2.3 lvalue and rvalue

- An **lvalue** can appear on the left side of an assignment operator
  - It is is an **object that can be modified**
- An **rvalue** appears on the right side of an assignment expression •
  - It is an expression that identifies something **temporary** that **can not be modified**

$$\underbrace{y}_{lvalue} = \underbrace{x+2}_{rvalue}$$

$$\underbrace{z}_{lvalue} = \underbrace{7}_{rvalue}$$

$$\underbrace{s}_{lvalue} = \underbrace{f(x)}_{rvalue}$$

$$\underbrace{7}_{rvalue} = \underbrace{z}_{lvalue} \Rightarrow ERROR$$

2.3.1 rvalue refrences

- It is illegal to assign a temporary rvalue to a reference variable:

```
int& r = x + 3 ; // Error
int i = 42
int &r = i ; // ok : r refers to i
```

- The following function call is illegal:

```
int f(int& n) { return 10 * n; }
x = f(x + 2);
```

- **C++ does have an rvalue reference**

```
int&& r = x + 3 ; // OK: note the two ampersands
int&& rr = i ; // error : cannot reference an lvalue
```

- The following function call is OK:

```
int g(int&& n) { return 10 * n;
x = g(x + 2);
```

נשים לב ש& ו && הם דברים שונים לגמרי, ולמשל ב*overloading* צריך לפצל:

```
void ref(int& n) { cout << "reference parameter: " << n << endl }
void ref(int&& n) { cout << rvalue reference parameter: " << n << endl }
```

שיעור 6

2.3.2 Move constructor

```
Vector::Vector (Vector && a)
: sz {a.sz}, elem a.elem } // move a.elem to elem
{
    a.sz = 0 ; // make a the empty vector
    a.elem = nullptr
}
vector fill( istream & is)
    vector res;
    for (double x; is>>x;)
        res.push_back(x)
    return res;
}
vector vec = fill(cin)
```

הסבר: בפונקציה $fill$ נבנה וקטור, ומחוזר ה $refernce$ שלו, אם לא היה את ה$move$ היתה מתבצעת העתקה של הוקטור, וגורמת לדליפת זיכרון

2.3.3 השמת הזזה

ההבדל בין בין השמת העתקה להשמת ההזזה הוא בחתימת הפנוקציה אם מקבלים & או &&

```
Vector & Vector ::operator=(Vector && a )
{
    delete [] elem ; // deallocate old space
    elem = a.elem ; // move a.elem to elem
    sz = a. a.elem = nullptr ; // make a the empty vector
    a.sz = 0 return *this ; // return a self reference
}
```

- If the caller passes an **rvalue** , the compiler generates code that invokes the **move constructor** or **move assignment** operator
- We want to avoid making a copy of the temporary

2.3.4 פעולות נדרשות למחלקות שתופסת משאבים

- $destructor$ נצרך כאשר המחלקה הקצתה משאבים, למשל:
  - המחלקה הקצתה זכרון למערך/וקטור וכד' - צריך לשחרר את הזכרון הזה
  - המחלקה פתחה קבצים, אז צריך לסגור אותם
  - $threads$ במצב נעול צריך לשחרר

– וכד'

- בד"כ כאשר למחלקה יש *destructor* , צריך לממש:

```
X( Sometype ); // ordinary constructor
X(); // default constructor
X( const X&); // copy constructor
X(X&&); // move constructor
X & operator=(const X&); // copy assignment
X & operator=(X&&); // move assignment
~ X (); // destructor
```

2.3.5 העמסת []

מימוש ראשון:

```
double operator[] (int i )
{
    return elem i
}
```

הבעיה היא שניתן רק לקרוא את את הערך ולא לכתוב, כלומר:

```
Vector v(10)
double x = v[ 2 ]; // fine
v[ 3 ] = x; // error, v[3] is not an lvalue
```

פתרון החזרת & . מימוש שני:

```
double& operator[ ](int n)
{
    return elem [n];
}
```

הבעיה היא מה עושים באובייקט שהוא *const* :

```
void f( const vector& cv) {
    double d = cv[1]; // Error , but should be fine
    cv[1] = 2.0 ; // Error , as it should be
}
```

- בראשון כיון ש*const* יודע לעבוד רק עם מתודות של *const* הקומפיילר "לא ידע" מה לעשות

- בשני אין לבצע שינוי ב$const$

פתרון, "מימוש כפול":

```
double& operator[] (int n); // for non const
const double& operator[] (int n) const ; // for const
```

2.3.6 העמסת +

- נגדיר אותו כ$nonmember\ function$ - ננמש אותו מחוץ למחלקה

  - הסיבה: לעיתים צריך לעשות $conversion$ בשביל להשתמש באופרטור ואם האופרטור חלק מהמחלקה יכול להיות שנתקע בהמרה למשל:

```
string s1 ,s2;
s1 = s + "abc";
s2 = "abc" + s;
```

  - אז ב $S2$ תתבצע המרה של s ל$cstring$, ו $s2$ שהוא $string$ לא יכול לקבל את ה"תוצר" הזה

- נגדיר את הפרמטרים כ $const$ כי אין סיבה לשנות אותם
- בד"כ נשתמש באופרטור התואם למימוש $+ =$
- דוגמת מימוש:

```
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs; // copy from lhs into sum
    sum += rhs ; // add rhs into sum
    return sum;
}
```

2.3.7 העמסת <<

- The **first** parameter of an **output** operator is a **reference** to a **nonconst** ostream object •
  - **nonconst** because writing to the stream changes its state.
  - **reference** because we cannot copy an ostream object
- The **second** parameter should be a **reference to const** to avoid copying and to avoid change
- To be consistent with other output operators, operator<< **returns its ostream parameter**
- output operators **should not print a newline** in order to let users print descriptive text along with the object on the same line

- IO Operators must be **nonmember functions** , the left hand operand cannot be an object of our class
- IO operators usually **need to read or write the nonpublic data members**, so they usually must be **declared as friends**

- דוגמת מימוש:

```
ostream operator<<( ostream & os , const V ector& vec
{
    os << '{';
    int n = vec.size
    if (n > 0 ) { // Is the vector non empty?
       os << vec [0]; // Send first element
       for (int i = 1 ; i < n; i++)
         os << ',' << vec i ];
    }
    os << '}';
    return os;
}
cout << vec 1 << vec 2 << endl
```

2.3.8 העמסת <<

- The **first parameter** is a **reference** to the stream from which it is to read
- The **second parameter** is a **reference to the (nonconst)** object into which to read, because the operator reads data into this object
- The operator usually **returns a reference** to its given stream
- Input operators must deal with the possibility that the input might fail
- we check once after reading all the data and before using those data

- דוגמת מימוש:

```
class Sales_data{
    std ::string bookNo;
    unsigned units_sold = 0;
    double price = 0;
    double revenue = 0.0 ;
istream  &operator>>(istream &is, Sales_data &item) {
    is >> item.bookNo >> item.units_sold >> item.price
}
```

2.3.9 "העמסת" איטרטורים

```
class Vector {
    int sz;       // the size
    double* elem; // a pointer to the elements
public:
    typedef double* iterator;
    typedef const double* const_iterator;

    iterator begin() { return elem; }
    const_iterator cbegin() const { return elem; }
    iterator end() { return elem+sz; }
    const_end cend() const { return elem+sz; }
// . . .
};
```

הערה: אם מבנה הנתונים הפנימי היה רשימה מקושרת, היה צריך למשש פונקציה שתחזיר את הערך במקום מסוים.

2.4 מימוש גנרי:

- We don't want just vectors of doubles, we want to specify the element type

```
template<typename T>
class Vector {
  T* elem; // elem points to an array of type T
  int sz;
public:
  explicit Vector(int s);
  T& operator[](int i);
  const T& operator[](int i) const;
};
template<typename T>
Vector<T>::Vector(int s) { . . . elem = new T[s]; . . .}
```

2.5 טיפול ב $Exceptions$

- ישנה התלבטות היכן לודא חריגות:
  - מצד אחד , מי שכותב את המחלקה עצמה מכיר את מגבלותיה הכי טוב, ולכן הגיוני שהוא יתמודד עם בעיות
  - מצד שני, מי שמשתמש במחלקה יודע הכי טוב, באילו תסריטים יכולות לצוץ בעיות ומה בדיוק הוא רוצה לעשות איתה, ולכן הגיוני שהוא יתמודד עם הבעיות
- לכן החליטו שמי שכותב מחלקה ידווח על תקלה, ומי שמשתמש יטפל בדיווח.
- לדוגמה $vector :: operator[]$ ו$out-of-range$:

```
double& Vector::operator[](int i)
{
```

```
    if (i < 0 || i >= size())
        throw out_of_range{"Vector::operator[]"}; return elem[i];
}
```

- מקובל לזרוק אובייקט - שהוא מחלקה עם הבעיה , במקרה שלנו בשם out_of_ragne
- תפיסת חריגות, בדרך המוכרת:

```
try { // exceptions are handled below
    // v[v.size()] = 7; // returns an undefined value
    v.at(v.size) = 7 // reports a bad index
} catch (out_of_range) { // oops: out_of_range error
    // ... handle range error ... }
```

בדוגמה , יש לשים לב שפניה מכוונת עם האופרטור [] תחזיר ערך לא מוגדר ולא תזרוק $exception$

2.5.1 $push\_back()$

בחשיבה ראשונית, הדרך ההגיונית היא להעתיק את כל הוקטור לשטח בזיכרון בגדול המקורי פלוס 1 , ואז להעתיק את כל הערכים, ובמקום האחרון לשים את החדש, הבעיה שזה מאוד לא יעיל ברצף של $push\_back()$, לכן מגדירים בסוף $space$, כלומר מקום ריק , ומעתיקים רק כאשר הוא מתמלא, לכן:

```
void Vector::reserve(int newalloc) {
    if (newalloc <= space)
        return;
    double* p new double[newalloc];
    for (int i=0; i<sz; ++i)
        p[i] = elem[i]; delete[] elem;
    elem = p;
    space = newalloc;
}
void Vector::push_back(double val) {
    if (space == 0)
        reserve(8);
    else if (sz == space)
        reserve(2*space);
    elem[sz] = val;
    ++sz;
}
```

3 הורשה

- הורשה נועדה להגדיר מחלקות שיש להם מכנה משותף
  - המחלקה המורישה נקראת מחלקת הבסיס
  - המחלקה היורשת נקראת המחלקה הנגזרת
- הורשת מימוש
  - מחלקת הבסיס מספקת למחלקה היורשת פונקציות ונתונים מוכנים
- הורשת ממשק
  - מאפשרת שימוש במחלקות היורשות השונות באמצעות הממשק של מחלקת הבסיס המשותפת
  - את המחלקות היורשות נקצה בזיכרון הדינמי באמצעות new
  - ניגש באמצעות מצביעים או משתני ייחוס למחלקת הבסיס

דוגמה:

```
class Employee {
    string first_name , family_name;
    Date hiring_date;
    int department;
};
class Manager : public Employee {
    list<Employee*> group;
    int level;
};
```

מבחינת הקצאה בזיכרון זה יראה כך:

כלומר הקצאת הזיכרון היא כמו של ה $employee$ בתוספת מה שנדרש בשביל $Manger$

דוגמת שימוש בפונקציה במחלקת הבסיס + תוספת של המחלקה היורשת:

```
class Employee {
public:
    void print() const;
// ...
};

class Manager:public Employee
{
public:
    void print() const;
// ...
};
```

```
void Manager::print() const
{
// print Employee info
    Employee::print();
// print Manager info
    cout << level;
// ...
}
```

3.1 פונקציה וירטואלית

הבעיה: אם יש לנו רשימה מעורבת של אובייקטים מעץ הירושה, וננסה לגשת לפונקציה משותפת להם (בדוגמה שלנו $print$) הפונקציה שתקרא תהיה לפי מחלקת הבסיס (כי כך הגדרנו את הרשימה) , והיינו רוצים שכל אובייקט ישתמש בפונקציה שלו, פתרון:

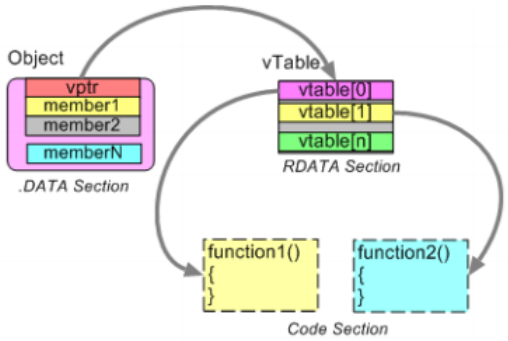
```
class Employee {
public:
    Employee(const string& name, int dept);
    virtual void print() const;
private:
    string first_name , family_name;
    short department;
};
```

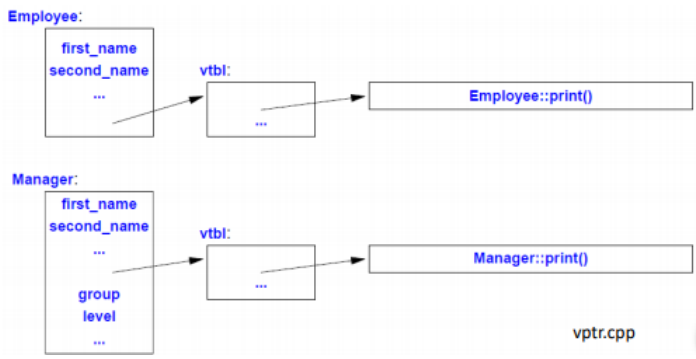
$virual$ מורה לקומפיילר לגשת לפונקציה של האובייקט

3.2 רב־צורתיות

איך זה עובד?

- בשביל מימוש ה $polymorphism$ הקומפיילר שומר את סוג האובייקט מכל אובייקט של $Employee$
- כאשר הוא פוגש $virtual\ function$ הוא ממיר אותו לאינדקס לטבלת $pointer$ לפונקציות - שנקרא $vtbl$
- $pointer$ בטבלה יכולה להצביע לפונקציה שהוצהרה ב $class$ עצמו או ב$class$ של מחלקת אב
- לכל מחלקה $member$ שהוא $pointer$ שמצביע על טבלת ה $vtbl$ שלה - בכך נרוויח שאם יש 100 אובייקטים מסוג מסוים, אין שכפול של ה $vtbl$, ו"כותבים" אותה פעם אחת.
- כאשר $virtual\ function$ נקראת , אז בעזרת ה$pointer$ הנ"ל הולכים ל$vtbl$ של ה $calss$, ומוצאים אותה בעזרת שם הפונקציה
- בציורים:





הערה : חשוב להגדיר את ה $destructor$ כ $virtual$ כי אחרת הקומפיילר ישתמש בכל פעם ב $destructor$ של מחלקת הבסיס ותהיה דליפת זיכרון

דוגמה

```
class Shape {
public:
    virtual Point center() const = 0; // pure virtual
    virtual void move(Point to) = 0;
    virtual void draw() const = 0;
    virtual void rotate(int angle) = 0;
    virtual ~~Shape() {} // virtual destructor is essential  since an object
                         // of a derived class may be deleted through a pointer
                         // to the base
```

עיגול היורש מצורה

```
class Circle : public Shape {
public:
    Circle(Point p, int rr); // constructor
    Point center() const { return x; }
    void move(Point to) { x = to; }
    void draw() const;
    void rotate(int) {} // nice simple algorithm
```

```
private:
    Point x; // center
    int r; // radius
};
```

סמיילי יורש מעיגול

```
class Smiley : public Circle {
public:
    Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }
    ~~Smiley() {delete mouth; for (auto p : eyes) delete p;}
    void move(Point to);
    void draw() const;
    void rotate(int);
    void add_eye(Shape* s) { eyes.push_back(s); }
    void set_mouth(Shape* s);
private:
    vector<Shape*> eyes; // usually two eyes
    Shape* mouth;
};
```

- כעת אם אנחנו רוצים להשתמש בפונקציה במחלקה היורשת צריך להוסיף את המילה *override* - לדוגמה *rotate*
- למה? כדאי למנוע מהמתכנת לעשות טעות של שגיאת הקלדה/פרמטר נוסף/פרמטר אחר ובכך לעשות *overloading* כלומר להוסיף פונקציה לעץ הירושה

```
class Smiley : public Circle {
    // . . .
    void move(Point to) override;
    void draw() const override;
    void rotate(int) override;
    // . . .
}
```

דוגמת הרצה - סיבוב מספר צורות

```
void rotate_all(vector<Shape*>& v, int angle)
{
    for (auto p : v) p->rotate(angle);
```

```
}
void user() {
    std::vector<Shape*> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v); // call draw() for each element
    rotate_all(v,45); // call rotate(45) for each element
    for (auto p : v)
        delete p;
}
enum class Kind { circle, triangle , smiley };
Shape* read_shape(istream& is) { // read shape header from is and find its kind k
switch (k) { // Kind k;
case Kind::circle: // read {Point,int}
    return new Circle{p,r};
case Kind::triangle: // read {Point,Point,Point}
    return new Triangle{p1,p2,p3};
case Kind::smiley: // read {Point,int,Shape,Shape,Shape}
    Smiley* ps = new Smiley{p,r};
    ps->add_eye(e1); ps->add_eye(e2); ps->set_mouth(m);
    return ps;
}
```

שיעור 8

3.2.1 *Dynamic_cast*

אם אנחנו רוצים להבין תוך כדי ריצה, מה הסוג (או הרמה) של אובייקט בהיררכיית הירושה , אז ישנם שני כלים ב $C++11$:

- *dynamic_cast* - מבצע *casting* בצורה מבוקרת מ*pointer* לאובייקט.
  - בכללי נשים לב שאנחנו עובדים עם *pointr*ים כאן.
- *type_id* - מחזיר את סוג האובייקט.
  - נציין ששימוש רב בפונקציות אלה מעיד על תכנון בעייתי

```
Shape* ps {read_shape(cin)};
    if (Smiley* p = dynamic_cast<Smiley*>(ps)) {
        // is a Smiley pointed to by p
    } else { // returns nullptr
```

```
    // not a Smiley, try something else
}
```

כלומר קיבלנו $Shape$= מחלקת בסיס, ובצענו בדיקה אם היא בכלל $Smiley$ = מחלקת יורשת

שמה ש"מפתיע" הוא שהיורשת "מצביעה" לבסיס (ראינו הרבה פעמים איך הבסיס מצביעה ליורשת)

סיכום $upcasting\ and\ casting$

- באופן כללי, מחלקה יורשת היא גם מחלקת בסיס ולכן לא צריך לעשות משהו מיוחד
- לעומת זאת המרה של $pointer$ של מחלקת בסיס למחלקת יורשת מצריכה פעולה מיוחד
  - הסיבה לכך שבמחלקה היורשת יש $data$ שלא קיים במחלקת האב

דוגמה ל $(casting)$:

$Programmer$ יורש מ $Employee$.

רק ל $Programmer$ יש פונקציה $coding()$, ומכאן נובעת השגיאה (שגיאת קומפליציה)

```
int main()
{
  Employee employee;
  Programmer programmer;
  Employee *pEmp =
    &programmer; // upcast
  Programmer *pProg = // down
   (Programmer *) &employee;
  pEmp->show_id();
  pProg->show_id();
  pEmp->coding(); // error
  pProg->coding();
```

4 קלט ופלט

4.0.1 קריאה וכתיבה ל $Stream$

נניח שיש לנו קובץ:

input file

morgan 2015552368 8625550123

drew 9735550130

lee 6095550132 2015550175 8005550000

התוכנה הנ"ל , תקרא מהקובץ ותצור וקטור של אנשים ומספרי הטלפון שלהם. כאשר בעזרת $istringstream$ אנחנו קוראים מהקובץ.

האובייקט הזה נותן לנו להתנהל עם $string$ כאילו הם מגיעים מקובץ/מקלדת

```
struct PersonInfo {
    string name;
    vector<string> phones;
};
vector<PersonInfo> getData(istream &is) {
    string line, word;
    vector<PersonInfo> people;
    while (getline(is, line)) {
        istringstream record(line);
        PersonInfo info; record >> info.name;
        while (record >> word)
            info.phones.push_back(word);
        people.push_back(info);
    }
    return people;
}
```

באותו אופן אפשר לכתוב בעזרת $ostringstream$

התכונה הנ"ל מקבלת את הוקטור שבנינו בחלק הקודם, ובודקת אם השם תקין. נשתמש ב $ostringstream$, כלומר קיבלנו מידע מהוקטור ,ואנחנו רושמים ל$stream$ שממנו ניתן לכתוב לקובץ או לפלוט למסך וכד'

```
ostream& process(ostream &os, vector<PersonInfo> people) {
    for (const auto &entry : people) {
        ostringstream formatted, badNums;
        for (const auto &nums : entry.phones) {
            if (!valid(nums)) {
              badNums << " " << nums;
```

```
            } else {
              formatted << " " << format(nums);
            }
        }
        if (badNums.str().empty())
            os << entry.name << " " << formatted.str() << endl;
        else
            cerr << "input error: " << entry.name << " invalid number(s) " << badNums.str() << endl;
        }
    return os;
}
```

איטרטורים ל$Stream$

```
istream_iterator<int> in_iter(cin); // read ints from cin
istream_iterator<int> eof; // end iterator
    while (in_iter != eof) // while there's input
                           // postfix increment returns the old value of the iterator
                           // we dereference that iterator to get the previous value
        vec.push_back(*in_iter++);
    we can rewrite this program as:
    istream_iterator<int> in_iter(cin), eof;
                           // construct vec from an iterator range
    vector<int> vec(in_iter, eof);  Using istream_iterator with the Algorithms
                           // generate the sum of values read from the input
    istream_iterator<int> in(cin), eof;
    cout << accumulate(in, eof, 0) << endl;
```

הרצאה 10 - חזרה על $stream$ והצגת תרגיל 3

שיעור 11

5 מחרוזות וביטויים רגולרים

נזכר במחרוזות בשפת C :

- בשפת C מחרוזת היא מערך של תווים שמסתיים בתוו שערכו 0
- בשפת C אין מחלקות אין פונקציות חברות ואין העמסת אופרטורים, וכדי לעבד מחרוזות משתמשים בפונקציות שמקבלות מחרוזת כפרמטר:

$$strlen() \ ,strcmp() \ ,strcat(), strcpy()$$

ב$C++$

- בשפת ++C נשתמש במחלקה $string$ שמאפשרת לעבד מחרוזות בצורה נוחה:

$$\text{s1 = s2 ; s1 += x ; s1 + s2 ; s1 == s2 ; s1 < s2 ; s.size() ; s.c\_str()}$$

- מימוש (שלנו) למחלקה:
  - יהיה $meber$ הוא מערך של תוויים שמתסיים ב 0
    - יעול: מחרוזות קצרות ישמרו בתוך האובייקט, ואורכות בזיכרון הדינמי
    - מחרוזת קצרה תשמר במערך ch
    - המשתנה $short\_max$ מכיל את הגודל המריבי (15) לשמירת מחרוזות בתוך האוביקט
    - המשתנה ptr יצביע על התו הראשון במחרוזת
    - גם כאן נקצה זיכרון עודף ליעל הוספה של תווים

```
class String {
private: static const int short_max = 15;
    int sz; // number of characters
    char* ptr;
    int space; // unused space on free store
    char ch[short_max+1]; // leave space for 0 (16)
};
```

מימוש בסיסי - עם כל מה שדגשים שלמדנו עד כה:

```
class String {
public:
    String(); //default constructor: x{""}
    String(const char* p); // C-style: x{"Euler"}
    String(const String&); // copy constructor
    String& operator=(const String&); // copy assignment
    String(String&& x); // move constructor
    String& operator=(String&& x); // move assignment
    ~String() { if (sz > short_max) delete[] ptr; }
    const char* c_str() { return ptr; } // C-style access
    int size() const { return sz; } // number of elements };
```

מימוש ה *constructor*

```
// Default constructor
String::String() : sz{0}, ptr{ch} {
    ch[0] = 0; // terminating 0
}
// C - style
String::String(const char* p) : sz{strlen(p)}, ptr{(sz<=short_max) ? ch : new char[sz+1]}, space{0}
{
    strcpy(ptr,p); // copy characters into ptr from p
}
```

5.0.1 *Regex*

מחרוזת המתארת תבנית של טקסט

נשתמש בפונקציות הבאות:

- *match_regex* פונקציה המנסה להתאים את הביטוי הרגולרי לכל המחרוזת
- *search_regex* פונקציה המנסה להתאים את הביטוי הרגולרי לחלק מהמחרוזת
- *replace_regex* פונקציה המחליפה מופע של הביטוי הרגולרי בטקסט אחר

דוגמת שימוש:

```
String input;
regex pat(<some pattern>);
while (true){
    cout<< "Enter text:" << endl;
    if(!(cin >> input)) break;
    if(regex_match(input, pat))
        cout<<"Match"<<endl;
    else
        cout<<"No Match"<<endl;
}
```

Regex meta-characters

- There are 12 punctuation characters that make regular expressions work their magic, they are called **meta-characters**
- Any regular expression that does not include any of the 12 metacharacters $()*+.?[\^{| simply matches itself

- If you want your regex to match them literally, you need to escape them by placing a backslash in front of them Thus,

  the regex: *\+\.\? matches the text *+.?

- Those backslashes may need to be doubled up to quote the regex as a literal string in source code (unless you use raw string): "*"
- Absent from the list are the closing square bracket], the hyphen -, and the closing curly bracket }
- The first two become metacharacters only after an unescaped [, and the } only after an unescaped {
- The rules about which characters are different inside a character class:
  - **dot** is a meta character outside of a class, but not within one
  - **dash** is a meta character within a class (between two characters), but not outside
  - **caret** has one meaning outside,
  - and another meaning if specified inside a class immediately after the opening [
- The notation [] - look for one mathces in list the inside []
  - [aeiouy] ⇔a or e or i or.....or y
- A hyphen (-) creates a range when it is placed between two characters • Match hexadecimal character: [a-fA-F0-9]
- A caret (^) negates the character class if you place it immediately after the opening bracket ("only first")
  - [^aeiouy] ⇔not an English vowel
  - [a^eiouy] ⇔ an English vowel or ^

Shorthand :

- $\backslash d$ matches a single digit
  - $\backslash D$ matches any character that is not a digit, and is equivalent to [^\d]
- $\backslash w$ matches a single word character, usually it is identical to [a-zA-Z0-9_]
  - $\backslash W$ matches any character that is not a word character
- $\backslash s$ matches any whitespace character - spaces, tabs, and line breaks
  - $\backslash S$ matches any character not matched by \s
- dot . matches any character except line breaks
- * (star) after a regex token means zero or more, example \d*
- + (plus) after a regex token means one or more, example \d+
- • A question mark ? after a regex token means zero or once
- The quantifier {n}, repeats the preceding regex token n number of times

- The quantifier {n,m}, repeats the preceding regex token n to m times

  example:

regex	matches
\d-\d	3-4,1-2
\w\w-\d\	Ab-12,12-34

Examples:
[A-Za-z_][A-Za-z_0-9]* an identifier in a programming language
0[xX][A-Fa-f0-9]+ C-style hexadecimal number
10{100} a googol (10^{100})
[A-Fa-f0-9]{1,8}h? 1-8 digit hexadecimal number with an **optional h** suffix
colou?r matches both colour and color
A*B+C? **matches** AAABBB, BC, B **does not match** AAA, AABBCC
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3} **matches** an IP Address

greedy and lazy matches

נסביר דרך דוגמה מעולם ה $html$

This is a <EM>first</EM> test

- על מנת לתפוס את $\langle EM \rangle$ כנראה שנכתוב את התבנית $\langle .+ \rangle$ אבל זאת תבנית שנקראת $\boldsymbol{greedy}$ והיא תתפוס את $< EM > first < /EM >$
  - בגלל שהתווים $*, +, \{n, m\}$ מבקשים את ההתאמה הכי גדולה שיש
- על לתפוס פחות נשים בסוף ? ובכך נהפוך את החיפוש ל $\boldsymbol{lazy}$
  - אצלנו: $\langle .+? \rangle$
  - או $\langle [\hat{} >] + \rangle$ (מתאים לדוגמה)
- דוגמאות נוספת:
  - $(ab)+ \Leftrightarrow$ - $greedy$ - לכן $abababa...$
  - $(ab)+? \Leftrightarrow$ - $lazy$ - לכן ab

שיעור 12

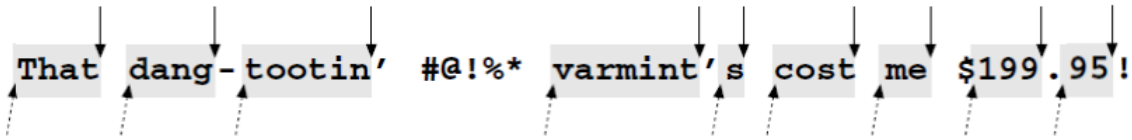
חיפוש

- The regular expression tokens ^, $ are called anchors
- They do not match any characters, instead they match at certain positions, effectively anchoring the regular expression match
- ^ (caret) matches only if it occurs at the **beginning of a string**
- $ (dollar) matches only if it occurs at the **end of a string**

- Examples:
  - ^cat$ a line that consists of only cat
  - ^$ an empty line
- Create a regex that matches **cat** in A **cat and a mouse,** but not in **category** or **bobcat** ⇒Place the word **cat** between two word boundaries

$$\backslash\text{b}\mathbf{cat}\backslash\text{b}$$

  - The regular expression token \b is called a word boundary, it matches at the start or the end of a word (In this example:)
    * The first \b requires the c to occur at the very start of the string, or after a non-word character.
    * The second \b requires the t to occur at the very end of the string, or before a non-word character



Match one of several alternatives

- The vertical bar or pipe symbol |, splits the regular expression into multiple alternatives
  - **Mary|Jane|Su**e matches Mary, or Jane, or Sue with each match attempt •
- The regular expression finds the leftmost match:
  - When you apply **Mary|Jane|Su**e to **Jane, Mary and Sue went to Mary's house**
  - the match Jane is found first
  - The match that begins earliest (leftmost) wins
- Each alternative is checked in a left-to-right order:
  - Jane|Janet matches Jane in Her name is Janet

Group parts of the match

- Improve the regular expression for matching Mary, Jane, or Sue by forcing the match to be a whole word
- Use grouping to achieve this with one pair of word boundaries for the whole regex, instead of one pair for each alternative
- **\b(Mary|Jane|Sue)\b** has three alternatives: Mary, Jane, and Sue, all three between two word boundaries
  - This regex does not match anything in **Her name is Janet**
- The alternation operator, has the **lowest** precedence of all regex operators
  - If you try **\bMary|Jane|Sue\b,** the three alternatives are **\bMary, Jane, and Sue\b**
  - This regex matches **Jane in Her name is Janet**

- Examples
- Nov(ember)? matches November and Nov (**greedy**)
- Feb(ruary)? 23(rd)? matches many alternatives
- \b(one|two|three)\b Find a line containing certain words:
- (\s|:|,)*(\d+) spaces, colons, commas followed by a number
- <HR(+SIZE *= *[0-9]+)? *> <HR SIZE=30>
- \$[0-9]+(\.[0-9][0-9])? Dollar amount with optional cents
- We have seen two uses for grouping parentheses:
  - to limit the scope of alternation
  - to group multiple characters into larger units to which you can apply quantifiers like question mark and star

נשים לב לשני סוגים סוגריים: 1) למקרה של *alternation* . 2) להגדיר לקבוצת תווים - פעולה מסוימת למשל ?

Capture parts of the match

- Create a regular expression that matches any date in **yyyy-mm-dd** format, and separately **captures** the **year, month**, and **day**
- A pair of parentheses isn't just a group, it's a capturing group
- Captures become useful when they cover only part of the regular expression, as in **\b(\d\d\d\d)-(\d\d)-(\d\d)\b**
- The regex **\b\d\d\d\d-\d\d-\d\d\b** does exactly the same, but does not capture
- Captures are numbered by counting opening parentheses from left to right
- There are three ways you can use the captured text:
  - match the captured text again within the same regex match
  - insert the captured text into the replacement text
  - The program can use the parts of the regex match
- Example:
  - $\underbrace{\hat{}\ ([-+]?[0-9]+\ \underbrace{(\backslash.[0-9]+)}_{\text{2 open parentheses}}\ ?)}_{\text{1 open parentheses}}\ \underbrace{([CF])}_{\text{3 open parentheses}}\ \$$
  - תבנית לטמפ'
  - מתחיל ב ^ ומסתיים ב $ ⇐ השורה הזו צריכה להכיל את הטמפ'

אמרנו שכל סוגריים נכנסים למשתנה, ויקרא $, ולכן סוגריים 1 יהיו $1 , וכו'...

עכשיו אם נרצה להשתמש ב *capture* , כיצד?

נסביר דרך דוגמת *date* "*magical*" :

- A date is magical if the year minus the century, the month, and the day of the month are all the same numbers
  -
- For example, 2010-10-10 is a magical date:
  - we first have to capture the previous text, then we match the same text using a back-reference

$$\backslash b\backslash d\backslash d\underbrace{(\backslash d\backslash d)}_{=\backslash 1}-\backslash 1-\backslash 1\backslash b$$

  - The (\d\d) matches 10, and is stored in **capturing group 1**
  - The **back-reference \1** matches the **10** of the **month** and **day**
- example 2 :
  - Match a pair of opening and closing HTML tags:

$$<([A-Z][A-Z0-9]*)[\hat{}>]*>.*?<\backslash 1>$$

  - • Checking for Doubled Words (the the):

$$\backslash b(\backslash w+)\backslash s+\backslash 1\backslash b$$

תוכנית לחיפוש בעזרת $regex$

$regex\_search()$ מוצא לפי התאמה **חלקית** ל $pattern$, בניגוד ל $regex\_match()$ שמחפש אחר התאמה **מלאה**

```
string input;
regex pattern(R"(\d+)");
smatch result;
while (true) {
    cout<<"Enter:"<<endl;
    if(!(cin >> input)) break;
    if(regex_search(input, result, pattern)) {
        cout<<"Match prefix: "<<result.prefix()<<endl;
        cout<<"Match string: "<<result[0]<<endl;
        cout<<"Match suffix: "<<result.suffix()<<endl;
    }
    else cout<<"No Match"<<endl;
}
```

התוצאות נכנסות למערך (במקרה שלנו $result$)

- כל התוצאה נכנסת למערך m - ונקבל את התוצאה ב $m[0]$
- $m.prefix$ כל המחרוזת עד ההתאמה
- $m.suffix$ כל המחרוזת מההתאמה
- $m[i]$ מחזירה את התשובה לפי סוגריים

```
string input {"x 1 y2 22 zaq 34567"};
regex pat {"(\w+)\s(\d+)"}; // word space number
string format {"{$1,$2}\n"};
cout << regex_replace(input,pat,format);
• The output is:
{x,1}
{y2,22}
{zaq,34567}
```

$sregex_iterator$

```
regex reg("([A-Za-z]+) \\1");
string target = "the the cow jumped over over the fence";
sregex_iterator reg_begin = sregex_iterator(target.begin(), target.end(), reg); sregex_iterator reg_en
for (sregex_iterator it = reg_begin; it != reg_end; ++it) {
    cout << "Substring: " << it->str() << ", ";
    cout << "Position: " << it->position() << endl; }
cout << "Found: " << distance(reg_begin, reg_end) << endl;
The output is:
Substring: the the, Position: 0
Substring: over over, Position: 19
Found: 2
```

סיכום

Special Characters

Regular Expression Special Characters

.	Any single character (a "wildcard")	\	Next character has a special meaning
[	Begin character class	*	Zero or more (suffix operation)
]	End character class	+	One or more (suffix operation)
{	Begin count	?	Optional (zero or one) (suffix operation)
}	End count	|	Alternative (or)
(	Begin grouping	^	Start of line; negation
)	End grouping	$	End of line

Repetition

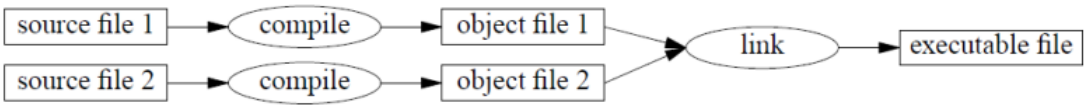
{n}	Exactly n times
{n,}	n or more times
{n,m}	At least n and at most m times
*	Zero or more, that is, {0,}
+	One or more, that is, {1,}
?	Optional (zero or one), that is {0,1}

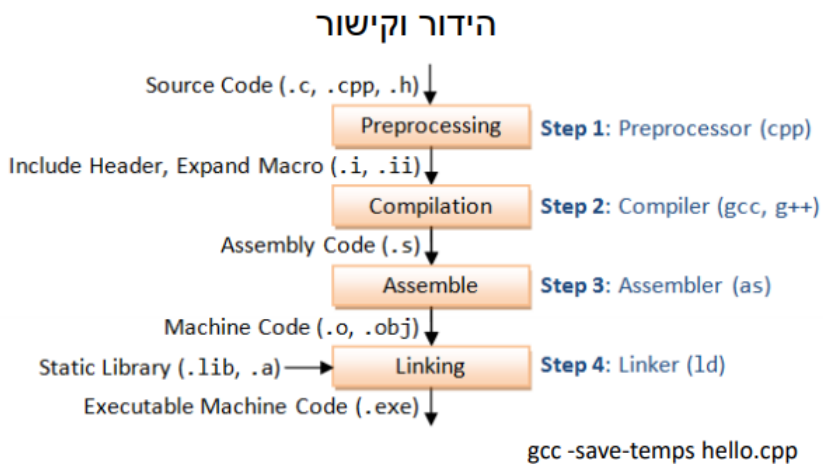
Character Class

\d	A decimal digit
\s	A space (space, tab, etc.)
\w	A letter (a-z) or digit (0-9) or underscore (_)
\D	Not \d
\S	Not \s
\W	Not \w

6 הידור וקישור

- נדבר על תהליך הקמפול
- הקומפיילר יוצר לכל קובץ $object\ file$ - הקובץ מכיל שפת מכונה, ולרוב יהיה בעל סיומת של $*.o$
- כל ה$object\ file$ (והספריות) מחוברות לקבוץ הרצה על ידי ה $linker$





- $preprocessing$:
  - כל מקום שכתוב $include$ הוא מחליף בכל הקובץ שעשינו לא $include$
  - מחליף $define$ בערך שלו
  - הערה: $include$ לא מעתיק את כל הספריה, אלא רק את החתימות של הפונקציה
- $Compilation$:
  - מתרגם $assmebly$
- $Assmbler$:
  - מתרגם לשפת מכונה (אפסים ואחדות)

שלושת השלבים הללו מתבצעים עבור כל קובץ- כעת הקובץ הוא עם סיומת $*.o$

- $Linking$:
  - מקשר את כל הקבצים + הספריות
  - יוצר קובץ $.ext$ (הרצה)

הערה: הפקודה הבאה מאפשרת לנו לראות את כל הקבצי הביניים:

gcc -save-temps hello.cpp

הצהרה והגדרה

- **הצהרה** על פונקציה או משתנה , רק אומרת לקומפיילר שתהיה פונקציה כזו - וניתן לכתוב זאת מספר פעמים
- **הגדרה** של פונקציה או משתנה, אומרת לקומפיילר להקצות מקום בזכרון, וזה נעשה רק פעם אחת

קבצי $header$

- נכתוב רק הצהרות, אחרת לאחר $include$ נקבל שגיאה

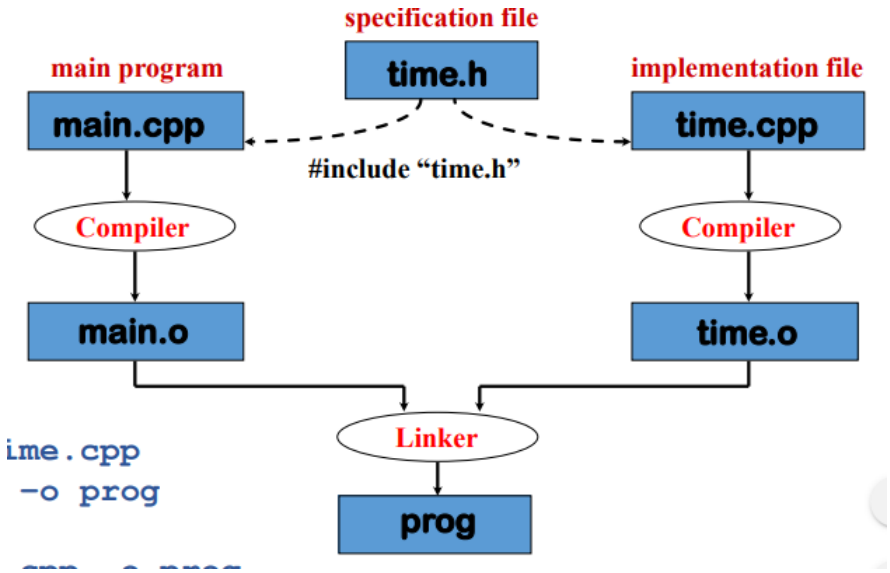
לדוגמה

int a; // =error

- נסייג: הצהרה על $class$ כן יכול להופיע ב$header$ ונוכל לעשות לו $include$ ממספר קבצים, לדוגמה

```
// s.h:
class S { int a; char b; };
// file1.cpp:
#include "s.h"
// file2.cpp:
#include "s.h"
```

- הצהרה על שימוש $Interface$ יבוצע ב $header$
  - מי שיעשה $include$ לקובץ יקבל גם את ה$interface$
  - ובנוסף, יקבל גם את הקובץ $.cpp$ שממש את ה$interface$
- דוגמה:



- הכלת קבצי כותרת ומניעת הכלות כפולות
  - לפעמים קבצי ה $header$ גם מכילים $include$ ויוצא שאנחנו עושים $include$לקובץ פעמים (וכפי שהראנו כל $include$ מעתיק את כל הקוד של הקובץ המצורף) על מנת למנוע בעיות שכאלה:

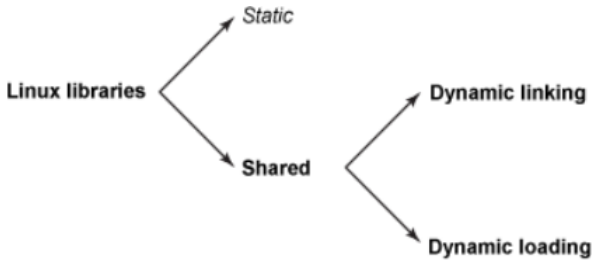
```
#ifndef SALESITEM_H
#define SALESITEM_H
// Definition of Sales_item class and related functions
#endif
```

- אפשרות אחרת:

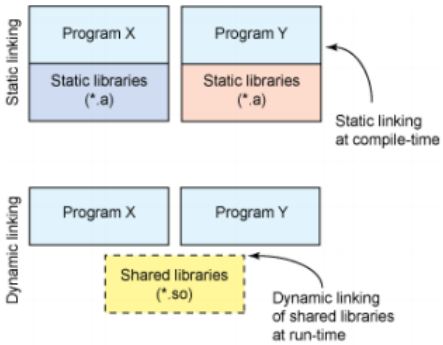
```
#pragma once
```

ספריות

משהו כתב עבורנו פונקציות שבשימוש רב. ישנן 2 סוגים:



- $Static$: מצורפות תמיד
  - מחבר קובץ עם סיומת $*.a$
  - גורם ל $linker$ לחפש סמלים (פונקציות) לא מוכרים בארכיונים, וכאשר הוא מוצא הוא מקשר אותם
  - חסרונות:
    - מכפיל את הקוד שכתוב - כי על כל קובץ צריך להעתיק את הספריה הסטנדרטית
    - כנ"ל בזמן ריצה , צריך כל הרצה לכל קובץ להעביר את הקוד הזה לזכרון
    - תיקון באגים: צריך לקמפל מחדש את כל התוכנה



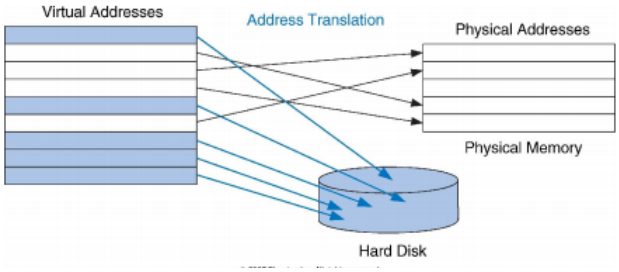
- פתרון: $Shared$: מצורפות על פי בקשת המשתמש ספריות דינאמיות
  - גם אחרי ה $linker$ הספריה לא מחוברת לתוכנה, אלא יש קובץ קטן שאומר לתוכנה ללכת לספריות שיושבות במערכת הפעלה, ובכך פתרנו את כל הבעיות.
  - ב$windows$ נפגוש אותן תחת השם DLL ובלינוקס so.
  - נטענות לזכרון כאשר מריצים את התוכנה
  - ניתן גם לומר למערכת ההפעלה לטעון ספריה ממש רק בזמן ריצה

שיעור 13

7 זיכרון

7.1 זיכרון מדומה $Virtual\ memory$

- מאפשר לכל תהליך להשתמש "בכל נפח הזכרון"
- חלק מהכתובת הוירטואליות נשמרות ב**זכרון הפיסי**
- הCPU מתרגם כתובות וירטואליות לפיזיות
- $Data$ שלא בזכרון הפיסי נמשכת מ$Hard\ drive$
- ה**זיכרון הפיסי** פועל כמו $cache$ לזכרון הוירטואלי



7.1.1 Locality in time and space

- **Locality in time** : keep recently accessed data in higher levels of memory hierarchy
- **Locality in space**: If data used recently, likely to use nearby data soon
  - If an element in an array is used, other elements in the same array are also likely to be used (array in loop)
  - bring nearby data into higher levels of memory hierarchy too

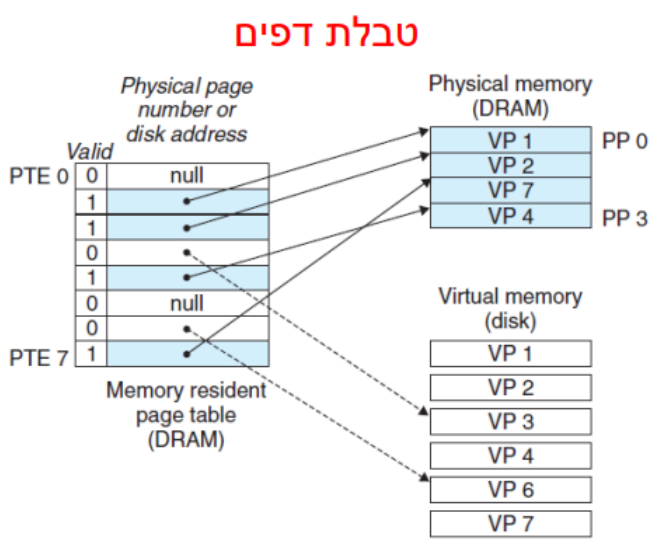
- במקרה ותוכנית רצה בסדר רנדומלי (מבחינת הזכרון) היא לא תרוויח מה $cache$ -כי כל פעם הוא יתמלא בגישה הרנדומלית החדשה
- ב$Cache$ אנחנו מניחים שישנה Locality in time and space כלומר שסביר שבפרק זמן מסוים ניגש לאותו מקום בזכרון:
  - לדגומה for(i=0;i<1000;++i) x[i] ; מסתמך על הנחה זו

7.1.2 כתובת פיזית ולוגית

- כל $byte$ בזכרון יש לו כתובת פיסית $PA-$ יחודית -
- הדרך הטבעית לגשת לכתובת בעבור ה CPU היא על ידי הכתובת הפיסית
- כאשר ה CPU ניגש לכתובת הוירטואלית (VA) הוא מתרגם אותה לכתובת פיסית
- ה $Hardware$ ל CPU נקרא MMU יודע לתרגם כתובת וירטואלית תוך כדי ריצה
  - לMMU יש טבלת כתובות, שמנוהלת על ידי המערכת הפעלה

7.1.3 תרגום כתובת לוגית לפיזית באמצעות טבלת דפים

- **הזכרון הוירטואלי** מחולק ליחידות גודל קבועות שנקראות virtual page
- באותו אופן **הזכרון הפיזי** מחולק ליחידות בגודל דומה שנקראות physical pages or page frame
- page table ממפה את הדפים הוירטואלים לפיזיים
- ה MMU קורא את ה page table בכל פעם שהוא מתרגם וירטואלית לפיזית, רצינו שתהיה יחידה מכנית שעושה זאת בשביל המהירות.
- לכל דף במרחב הכתובות הוירטואלית, יש page table entry (PTE) :
  - valid bit -
    - אם הוא במצב set סימן שהכתובת בזכרון היא התחלה של **physical page**
    - אם הוא במצב $not\ set$ ויש $null\ adress$ סימן שה virtual page לא בזכרון, והוא לא תופס מקום על ה $disk$
    - אחרת, הכותבת מצביע להתחלה של virtual page על ה $disk$, כלומר יש $page$ בזכרון, אבל **אינו בזכרון הפיסי**



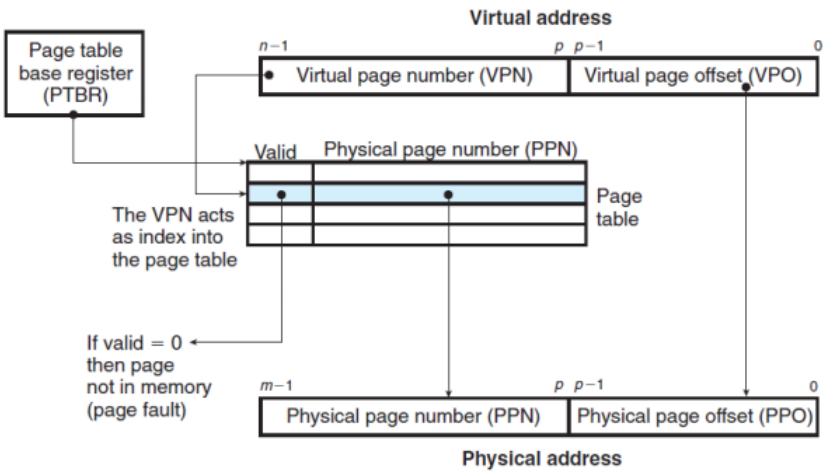
נראה את תהליך תרגום כתובת, בעזרת דוגמה:

- נניח שאורך כתובת היא $32bit$
- נניח שגודל דף $\text{Page offset} = 12\mathbf{bits} \Leftarrow 4KB = 2^2 \cdot 2^{10} = 2^{12}\boldsymbol{Byte}$

$$\underbrace{\boxed{20}\boxed{12}}_{32\ \text{bit}}$$

כלומר 20 הביטים הראשניים יפנו אותנו לדף הנכון, וה12 הנוספים יכוונו אותנו למקום בתוך הדף.

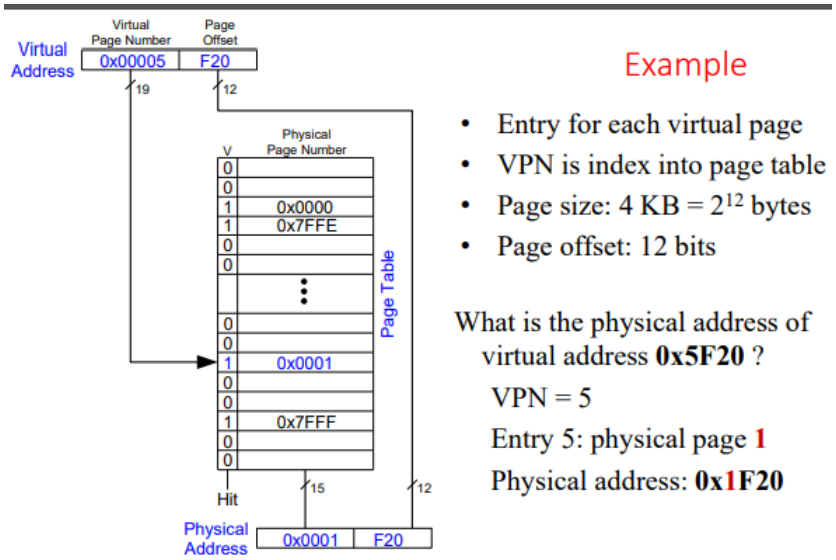
תרגום כתובת לוגית לפיזית באמצעות טבלת דפים



ניתן לראות שגודלו ה$offset$ זהה, כיון שגודל ה $page$ זהה בפיסי ובוירטואלי.

תחילת הכתובת מתורגמת על פי טבלת הדפים - ונדע לגשת לשורה הנכונה על פי ההתחלה של הכתובת הוירטואלית

דוגמה:



12 הביטים האחרונים בכתובת, הם: $F20$ (ה $offset$)

התחילית היא $0x00005$ $\Leftarrow$ דף לוגי 5

ע"פ הציור בטבלת הדפים $\Leftarrow$ בשורה 5 מופיע: $0x001$, ולכן כאשר נרכיב את הכתובת הפיסית:

$0x0001$	$F20$

- דוגמאות לחישובים:

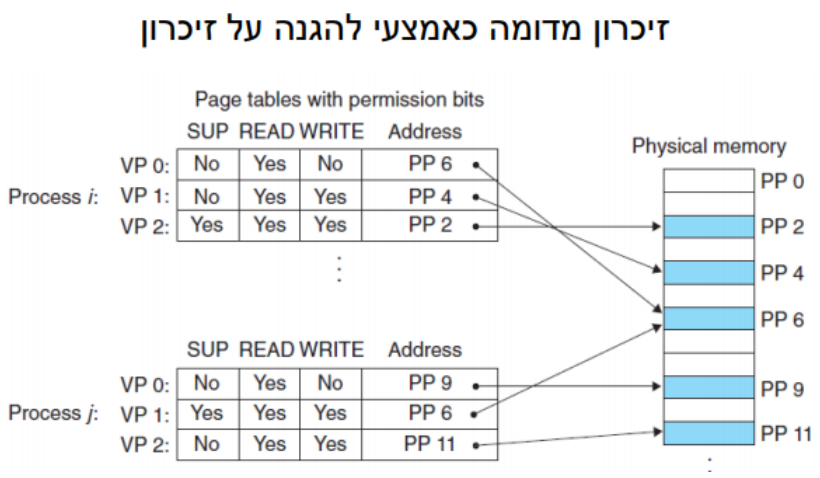
נניח שכתובת וירטואלית היא 32 ביט, קבע את מספר הביטים למספר הדף הוירטואלי , ומספר הביטים ל$offset$:

- עבור גודל דף $1KB$:
  - $1KB = 2^{10} \Leftarrow$ ה$offset$ של 10 ביטים, $page\ number$ 22 ביטים
  - עבור גודל דף $2KB$:
    - $2KB = 2^{10} \cdot 2 = 2^{11} \Leftarrow$ ה$offset$ של 11 ביטים, $page\ number$ 21 ביטים

7.1.4 זיכרון מדומה כאמצעי להגנה על זיכרון

ניתן להוסיף הרשאות, בטבלת הדפים, למשל:

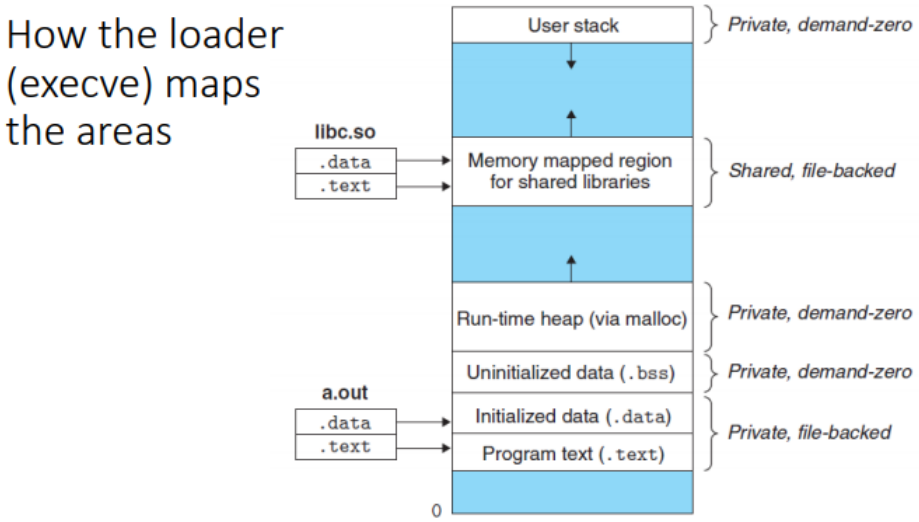
- קוד של תוכנית נתיר שיהיה לקריאה בלבד
- קוד של מערכת ההפעלה - לקריאה בלבד
- נרצה שתהליך לא יוכל לקרוא/לכתוב לזכרון של תהליך אחר - מתבצע על ידי הגדרת אזורים שונים לכל תהליך
  - ניתן להוסיף ביט PTE לבקרת גישה עדינה יותר
  - ביט SUP - ביט שמצביע האם תהליך חייב לרוץ על ה $kernel$ בשביל לגשת לזכרון הנ"ל.



יתרון נוסף: למקרה ורוצים זכרון משותף נוכל להגדיר זאת באופן לוגי בטבלת הדפים

- יתרונות:
  - שימוש יעיל בזכרון על ידי הפיכותו למעין $cache$
  - נותן לנו את האפשרות להשתמש בזכרון כאילו יש לנו יותר זכרון ממה שיש לנו בפועל
  - הגנה על מידע בין תהליכים
  - העבודה של ה$complier$ וה $linker$ יותר פשוטה, כי הם מתייחסים לאותן כתובות
- פסיקת דף:
  - במקרה שדף כבר $invalid$ כלומר אז ה $DRAM$ יקבל $cache\ miss$ ידוע גם כ $page\ fault\ exception$
  - בשלב הזה ה MMU יוצר פסיקה $(interrupt)$ שמודיע למערכת ההפעלה, שצריך ללכת להביא דף מהזכרון (HW)

7.2 הקצאת זכרון דינמית



מלמטה למעלה:

- $text$ - קוד התוכנית
- $data$ - משתנים גלובלים לתוכנית
- $heap$ - $malloc, new$ - גדל כלפי מעלה . בזה נתעמק עכשיו
- $stack$ - גדל כלפי מטה
  - הספריות בין ה $heap$ ל$stack$

7.2.1 הקצאת זכרון דינמית

- הקצאת זכרון דינמית מקצה אזור ב**זכרון הוירטואלי** וידוע בשם $heap$
- ה $allocator$ שומר על ה $heap$ כאוסף $block$ים בגדלים שונים
- כל בלוק הוא חתיכה רציפה של זכרון בשימוש או פנוי
- ישנו משתנה brk שמצביע על קצה ה$heap$
  - $Explicit\ allocators$ - דורש שישום באופן מפורש ישחרר זכרון
  - $Implicit\ allocators$ - $garbase\ collectos$ ה $allocator$ נדרש להבין מתי לשחרר בלוק

7.2.2 malloc and free

- $malloc()$ - מחזירה פוינטר ל $block$ עם הזכרון המבוקש
  - הבלוק ביחידות של $8 - byte$
- $free$ משחרר את הזכרון
- $*sbrk()$ - מגדיל את הערמה , לרוב לא נשתמש כי ה $malloc$ יעשה זאת לבד

- הפונקציות הנ"ל לא "מנקות" את הזכרון
- ב$C++$ נשתמש ב:
  - new - הקצאת זיכרון דינמית + ואתחול
  - $delete$ - שחרור וניקוי זכרון
- בגלל שהם מוגדרים כאופרטורים הקומפיילר מבטיח לקרוא לconstructors and destructors

```
MyType* fp = new MyType[100];
```

- השורה הזו תקרא ל constructor של כל אובייקט $\Leftarrow$ צריך להיות מוגדרת $default$ כי זה הולך להתבצע לכל המערך

```
delete [] fp;
```

- קורא ל destructor של כל איבר במערך

```
delete fp;
```

- קורא רק ל destructor של האיבר הראשון במערך

7.2.3 בעיות בניהול זיכרון ידני

- memory leak
- Double free - באמצע משהו השתמש ואז נדרס המידע
- use-after-free

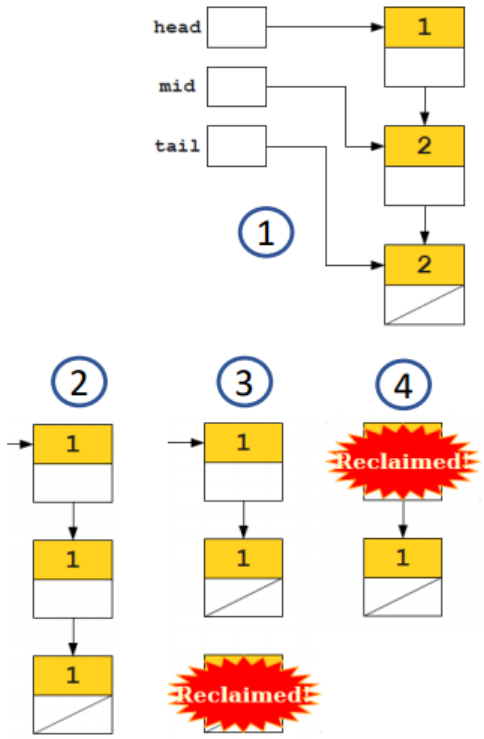
Automatic Management - Reference Counting

- איך זה עובד?
  - לכל אובייקט יש $counter$ שסופר כמה מצביעים יש עליו :
    * עולה כאשר הוגדר $refernce$ חדש למשל בכניסה לפונקציה
    * יורד כאשר $refernce$ נמחק למשל ביציאה מפוקציה
    * נמחק כאשר ה$counter = 0$

דוגמה:

Reference Counting Example

```
class LinkedList {
    LinkedList next;
}
void f() {
LinkedList *head = new LinkedList;
LinkedList *mid = new LinkedList;
LinkedList *tail = new LinkedList;
head->next = mid;
mid->next = tail; (1)
mid = tail = null; (2)
head->next->next = null; (3)
head = null; (4) (5) all reclaimed
}
```



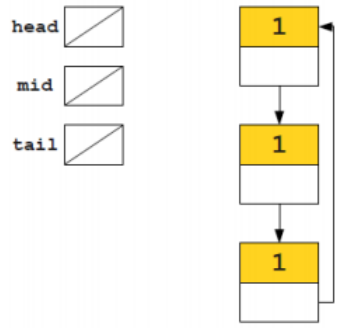
הסבר:

בהתחלה שלושה מצביעים לשלוש רשימות מקושרות + קשרים בינהם (1) ואז ביטול $mid, tail$ (2) , ביטול המצביע לרשימה המקושרת השלישית $\Leftarrow couunter = 0$ מחיקת $linklist\_3$ (2) באופן דומה ב (3) , באופן דומה ל (4)

7.2.4 Reference Cycles

- אובייקטים שמצביעים אחד על השני במעגל, יכול להווצר מצב שהם לא ישיגים ולא ימחקו כי $counter > 0$

```
LinkedList *head = new LinkedList;
LinkedList *mid = new LinkedList;
LinkedList *tail = new LinkedList;
head->next = mid;
mid->next = tail;
tail->next = head;
head = null;
mid = null;
tail = null;
```

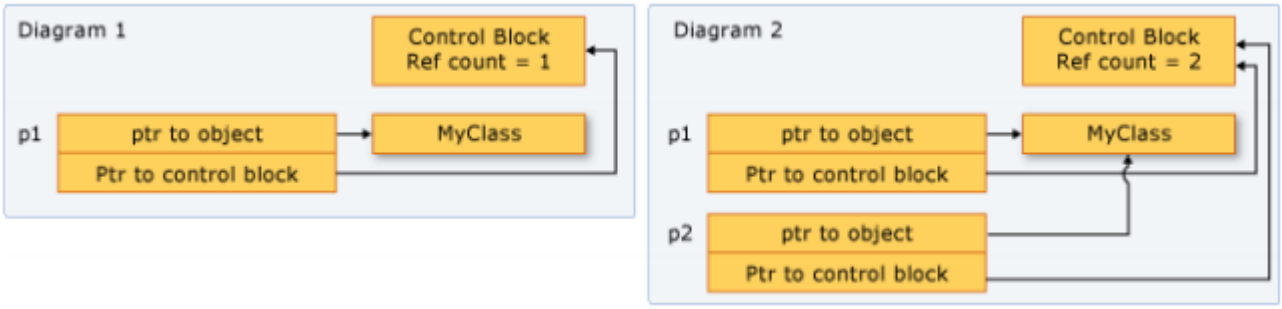


Shared_ptr

- הוא בעצם האובייקט שדיברנו עליו שנותן את האפשרות ל$counter$
- $shared\_ptr$ הוא $class$ עם $*$ ו $->$

- shared_ptr **constructor increases** the reference count

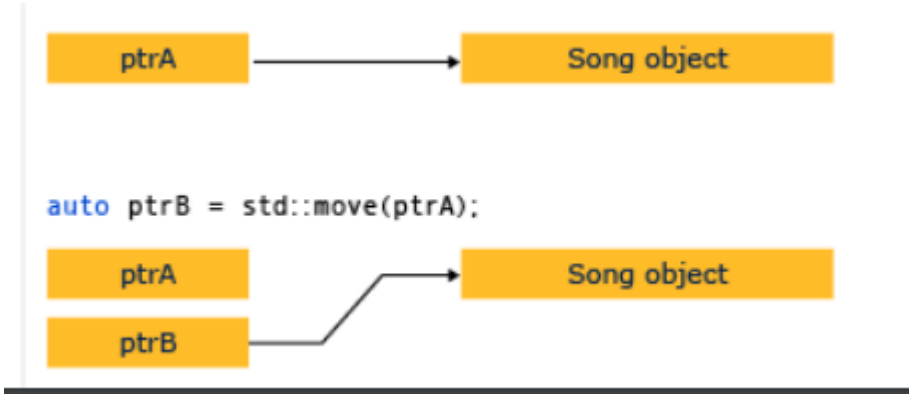
- shared_ptr **destructor decrements** the count and **if it reached 0 deletes**
- shared_ptr **can be copied, passed by value, and assigned**



נשים לב ש *control block* "חיצוני" כך אפשר לנהל את הספירה

7.2.5 unique_ptr

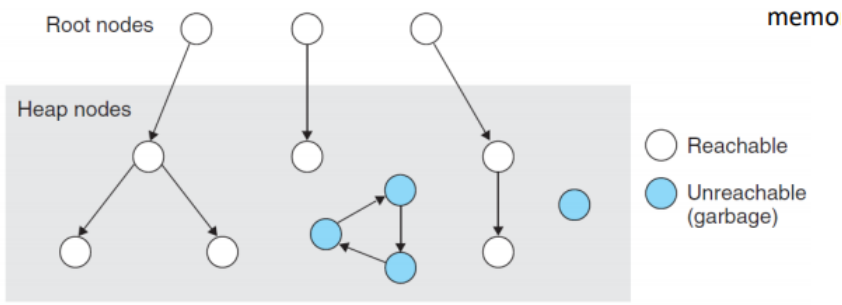
- אותו רעיון כמו ה *shared* רק שהוא יחודי, ולכן אין *counter*
- לכן יותר קטן ומהיר, ולא ניתן להעתיק או להעביר *by value*
- ניתן לעשות לו *move*



7.2.6 Garbage Collector

- נאמר שאובייקט *reachable* אם יש לו *refernce* כלשהו בתכונית
- ישנם אובייקטים *reachable* ועדיין לא נשתמש בהם מעולם (כמו הרשימה המעגלית ממקודם), או סתם הצבעות לא שימושיות
- באופן אטומטי ניתן להגדיר אובייקטים כ*unreachable* וזו פעולת ה Garbage Collector
- איך זה נעשה?
  - Garbage Collector מסתכל על הזכרון כגרף מכוון
  - הגרף מחולק ל
    - root set : איברים שעל ה*stack* או בזכרון הגלובלי

* Heap set : מתאים לבלוקים שהוקצו בערמה
  - אם אין מצביע מאחד מהם, סימן שזה זכרון שדלף



- פעולות ה garbage collector (בע"פ מפנחס):
  - מעבר ראשון עובר על הזכרון הגלובלי ועל ה*stack* ומסומן את הכתובת שהן מצביעות עליהן
  - שלב שני עובר על ה*heap* ובודק את הכתובות המוצבעות, ואם יש הצבעות בתוך ה *heap*
    * בכל אחת מהבדיקות אם יש התאמה מסמן כזכרון פעיל
  - מעבר שני , עובר בשנית על כל הזכרון - גלובלי, *stack* , *heap* - כל מה שלא מסומן כפעיל מוחק.

מהמצגת:

- block **q** is **reachable** if there exists a directed path from any root to q
- The path may include a directed edge **p** $\rightarrow$ **q** where some location in **block p** points to some location in **block q**
- **Unreachable** blocks correspond to **garbage**, they can never be used by the application
- A **Mark and Sweep** garbage collector consists of two phases:
  - A **mark phase**, which marks all reachable and allocated descendants of the root nodes
  - A **sweep phase**, which frees each unmarked allocated block
- Typically, one of the spare low-order bits in the **block header** is used to indicate whether a block is marked or not

Mark and Sweep

```
typedef void *ptr;
ptr isPtr(ptr p): If p points to an allocated block,
returns a pointer to that block

void mark(ptr p) {
   if ((b = isPtr(p)) == NULL)
     return;
   if (blockMarked(b))
     return;
   markBlock(b);
   len = length(b);
   for (i=0; i < len; i++)
     mark(b[i]);
   return;
}

void sweep(ptr b, ptr end) {
   while (b < end) {
      if (blockMarked(b))
         unmarkBlock(b);
      else if (blockAllocated(b))
         free(b);
      b = nextBlock(b);
   }
   return;
}
```