# פונקציות

## ברירת מחדל

ניתן להגדיר בהצהרה של פונקציה ערך ברירת מחדל בפרמטרים. void foo(int n=5).

## const

יש שלושה מקומות בהם ניתן להוסיף את המילה const בהצהרה על פונקציה:

1. **בתחילת ההצהרה** - משמע הערך המוחזר קבוע. מיועד לפונקציות שהערך המוחזר מהם נכנס לתוך משתנה קבוע.
2. **על הפרמטרים** - משמע שהפרמטרים המתקבלים הם קבועים ולא ניתן בתוך הפונקציה לבצע עליהם השמה.
3. **בסוף ההצהרה** - רק בתוך שיטות של מחלקה. לא מאפשר לעשות שינוי על האובייקט הנוכחי שממנו הופעלה השיטה (this).

## העמסת פונקציות

ב-C++ יש מנגנון שנקרא "העמסת פונקציות" (Function Overloading). מנגנון זה מאפשר לנו להגדיר כמה פונקציות עם אותו שם אך עם ארגומנטים שונים. התועלת מכך היא שנוכל לקרוא לפונקציות בשמות קצרים וברורים כך שלא נצטרך להמציא שם חדש לכל מקרי קצה של פעולה מסוימת. כאשר מוגדרות לנו שתי פונקציות כאלו.

בקריאה לפונקציה הקומפיילר בוחר איזו מהפונקציות נכונה. כיצד הוא עושה זאת? בכמה שלבים:

1. מציאת כל הפונקציות עם השם המתאים.

2. מתוך קבוצה 1, מציאת כל הפונקציות עם מספר הפרמטרים המתאים.

3. מתוך קבוצה 2, מציאת הפונקציות עם סוג הפרמטרים **המתאים ביותר**.

בשלב השלישי, אם הקומפיילר לא מוצא בדיוק את הפונקציה שמקבלת ארגומנטים מסוג הפרמטרים שהכנסנו, הוא יעשה עליהם casting. לכן יכול להיות מצב שבו יתקבלו שתי פונקציות מתאימות. במצב זה הקומפיילר יחזיר שגיאת ambiguous (רב משמעות).

לדוגמה: בפונקציות הבאות אם נקרא לפונקציה כך power(2, 3.5) נקבל שגיאת קומפילציה, כי בשני המקרים צריך לבצע casting למספר אחד לכן רמת ההתאמה היא שווה.

int power (int a, unsigned int b) {

cout << "power of ints" << endl;

return b==0? 1: a\*power(a,b-1);

}

double power(double a, double b) {

cout << "power of reals" << endl;

return exp(b\*log(a));

}

המסקנה היא שצריך מאד להיזהר בהעמסת פונקציות, במיוחד כשמדובר בפרמטרים מספריים.

## ביטויי למדא

ביטוי למדא הוא ביטוי היוצר אובייקט שיש לו סוגריים עגולים (). השימוש העיקרי בביטויי למדא הוא כאשר יש פונקציה f שמקבלת כפרמטר פונקציה אחרת g. במקרה זה נוכל להעביר כפרמטר ל-f או שם של פונקציה כמו שאנחנו מכירים, או אובייקט מחלקה שיש לה אופרטור סוגריים עגולים, או ביטוי למדא. האפשרות השלישית היא במקרים רבים הנוחה ביותר, שכן מאפשרת לנו ליצור כמעין פונקציה זמנית, תוך כדי הקוד, שאינה כתובה במקום מיוחד.

כדי ליצור ביטוי למדא נרשום סוגריים מרובעים ריקים, לאחר מכן סוגריים עגולים שבתוכן הפרמטרים לביטוי למדא, ולאחר מכן סוגרים מסולסלים המכילים את הפקודות והערך חזרה מהביטוי למדא. לדוגמה, ביטוי למדא המקביל לפונקציה add.

[](int x, int y){return x+y;}

# העמסת אופרטורים

## הגדרה

בשפת ++C גם אופרטור הוא פונקציה, ולכן אפשר לבצע העמסה על רוב האופרטורים המובנים בשפה, כדי שנוכל להפעיל אותם על פרמטרים שונים שנרצה, וכן לקבוע מה יתבצע כשאופרטורים אלו יופעלו. כמו כן, ניתן להגדיר אופרטורים חדשים שאינם מובנים בשפה.

העמסת אופרטורים מאוד דומה להצהרה על פונקציה, אלא שבשם הפונקציה נכתוב את המילה השמורה 'operator' ולאחר מכן את הסמל של האופרטור שאותו אנו מגדירים. כמו בפונקציות, יש להגדיר פרמטרים שהיא מקבלת בסוגריים וכן ערך חזרה. לדוגמא, כדי להעמיס את האופרטור '+' כדי שיופעל בין משתנים מסוג Point:

Point **operator**+(const Point& p1, const Point& p2) {

return Point(p1.a+p2, p1.b+p2.b);

}

המטרה העיקרית בהעמסת אופרטורים היא נוחות וקריאות הקוד, הרי הקוד a = b + c קריא יותר למתכנת מאשר a.set(add(b, c)). לכן כשמעמיסים אופרטורים, חשוב לשים לב שהערך המוחזר תואם למשמעות של האופרטור כפי השימוש המקובל באופרטור. נביא לכך דוגמאות בהמשך.

חשוב לשים לב שגם שמעמיסים אופרטורים **סדר הפעולות המוכר נשמר**. לדוגמא, תמיד אופרטור כפל יבוא לפני אופרטור חיבור. על סדר הקדימות בין כל האופרטורים אפשר לקרוא [כאן](https://en.cppreference.com/w/cpp/language/operator_precedence). בנוסף, העמסת אופרטורים **אינה קומוטטיבית**, כלומר אם לדוגמא הגדרנו אופרטור כפל בין string ל-int, אז יש גם להגדיר אופרטור כפל בין int ל-string. המעבר לא יעשה אוטומטית.

את האופרטורים '.' , '::' , ':?' , 'sizeof', ואופרטורים נוספים לא ניתן להעמיס. אפשר גם לדרוס אופרטורים קיימים שכבר הוגדרו בספרייה הסטנדרטית, אך לא ניתן לדרוס אופרטורים על משתנים פרימיטיביים, כלומר אחד מהפרמטרים חייב להיות לא פרימיטיבי.

הקריאה לאופרטור נעשית דרך הסמל בלבד, אמנם נוכל לקרוא לפונקציית האופרטור באמצעות שמה המלא. לעיתים לא תהיה לנו ברירה אלא לציין שם מלא זה, למשל כשנרצה לקבל מצביע לפונקציית האופרטור.

## מימוש מחוץ למחלקה ובתוך מחלקה

כל האופרטורים שניתן להעמיס עליהם מתחלקים לשניים - כאלה שניתן להעמיס עליהם רק כשיטות בתוך מחלקה, וכאלה שניתן להעמיס עליהן גם מחוץ למחלקה. ראה [כאן](https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B). ההיגיון פשוט, אופרטורים שבמשמעות המקורית שלהם הופעלו על אופרנד שהוא lvalue (כלומר אובייקט הנמצא בזיכרון), ניתן להגדיר רק בתוך המחלקה. ואילו אופרטורים שלא שינו את האופרנדים שלהם ולא עבדו עם הכתובת שלהם, ניתן להגדיר גם מחוץ למחלקה.

במימוש מחוץ למחלקה, לאופרטור אין אובייקט שהוא מופעל עליו (אין this), לכן כל הערכים שהאופרטור מופעל עליהם חייבים להתקבל כפרמטרים, כמו בדוגמא מסעיף קודם. ברוב האופרטורים, כמו אופרטורים אריתמטיים, לא נרצה לשנות את הפרמטרים שהתקבלו, לכן הפרמטרים יהיו משתנים לוקליים או משתני רפרנס קבועים, אך לא יהיו מצביעים או רפרנס פשוטים. כמו כן, מסיבה זו ערך החזרה יהיה משתנה מקומי ולא רפרנס. אם רוצים שלאופרטור יהיה גישה למשתני מחלקה פרטיים נצרך להגדיר את האופרטור כפונקציה friend של המחלקה.

במימוש בתוך המחלקה, כל אופרטור שנעמיס מופעל על האובייקט this. לכן בכל מצב האובייקט עצמו מהווה את הפרמטר הראשון לאופרטור. במקרה זה, אם האופרטור משנה את האובייקט נחזיר רפרנס ל-this כדי שנוכל להפעיל על האובייקט המוחזר עוד אופרטורים. ואם לא נרצה לשנות נחזיר משנה מקומי ונגדיר את האופרטור כ-const.

class Point {

...

Point operator+(const Point& p) const {

return Point(this->a + p.a, this->b + p.b);

}

};

אם מגדירים אופרטור בין טיפוס המחלקה כפרמטר שמאלי לטיפוס אחר כפרמטר הימני, אזי כיוון שהעמסת אופרטורים אינה קומוטטיבית לא נוכל להפעיל את האופרטור בכיוון ההפוך, שהטיפוס האחר הוא פרמטר שמאלי וטיפוס המחלקה הוא פרמטר ימני. לשם כך יש גם להגדיר אופרטור שהוא פונקציה friend של המחלקה שעושה זאת.

## סוגי אופרטורים

* + **אופרטורים אריתמטיים** - חיבור, חיסור, כפל, השוואה, וכו'. מוגדרים על מספרים שלמים וממשיים. אנחנו יכולים להעמיס אותם גם במחלקות שאנחנו בונים, המייצגים עצמים מתמטיים מורכבים יותר. למשל: מספרים מרוכבים, מטריצות, פולינומים וכו'. בספריה התקנית הם הורחבו גם למחרוזות.
  + **אופרטורים אונאריים** - כמו '++', '--', '!'. ניתן להעמיס רק בתוך המחלקה והם לעולם לא מקבלים פרמטרים. בדרך כלל מופעלים בצד שמאל של האובייקט. כדי שיופעלו בצד ימין יש להכניס כפרמטר מספר int שישמש כדגל לכך שהאופרטור האונארי הוא בצד ימין (postfix).

Complex& operator++() // ++object

const Complex operator++(**int**) // object++

עבור אופרטורים אונאריים בצד שמאל, בדרך כלל נרצה שיחזירו רפרנס לאובייקט, שכן לאחר שעשינו פעולה על האובייקט והחזרנו אותו, נרצה לשרשר עוד פעולות לאחר מכן. אמנם עבור אופרטורים אונאריים בצד ימין נרצה שיחזירו ערך const זמני של האובייקט, מפני שהערך המוחזר מהם הוא לא lvalue, ולא ניתן לשרשר עליהם עוד פעולות.

* + **אופרטורי השוואה** - <, >, <=, >=, ==. מקבלים שני פרמטרים ומחזירים בדרך כלל ערך בוליאני.
  + **אופרטורי השמה** - אופרטור '=' הקומפיילר מוסיף אוטומטית לכל מחלקה. לוקח את הפרמטר בצד ימין ומציב אותו בפרמטר בצד שמאל. חשוב שאופרטור זה יחזיר רפרנס כדי שניתן יהיה לבצע פעולות בשרשרת. אופרטור '+=' מגדיל את הפרמטר השמאלי שלו אבל גם מחזיר את הסכום (שהוא הפרמטר השמאלי אחרי ההשמה).
  + **אופרטור סוגריים מרובעים []** - מוגדר לגבי מערכים בסיסיים. מקובל להרחיב אותו גם למבני נתונים שאנחנו בונים. האופרטור [] יקבל כפרמטר מספר שישמש כאינדקס למבנה הנתונים, ויחזיר את האיבר באותו אינדקס - [i]. אם נרצה שהאופרטור רק שיחזיר את האיבר נגדיר את הערך המוחזר const, אך אם נרצה גם לשרשר עליו פעולות יחזיר רפרנס.
  + **אופרטור סוגריים עגולים ()** - עבור אובייקטים המתפקדים כמו פונקציות, אפשר להעמיס את האופרטור () כך שיקבל ערכים x1, x2,…, xn, ויחזיר את הערך של האובייקט עבור אותם משתנים. אפשר גם להגדיר כמעין בנאי באמצעות אופרטור זה.

## אופרטורי זרימה << >>

מוגדרים במקור (בשפת סי) על מספרים שלמים, אבל בשפת ++C העמיסו אותם לזרמי קלט ופלט. גם אנחנו יכולים להרחיב אותם כדי לכתוב ולקרוא מחלקות שאנחנו בונים. חשוב להעמיס אופרטורים אלו כפונקציה friend בתוך מחלקה כדי שיהיה אפשר להפעיל אותם גם בלי ליצור אובייקט.

friend ostream& operator**<<**(ostream &output, const Point &p) {

output << "(" << p.x << ", " << p.y << ")";

return output;

}

friend istream& operator**>>**(istream &input, Point &p) {

input >> p.x >> p.y;

return input;

}

## אופרטור גרשיים ""

נקרא גם אופרטור הסיומת (suffix operator). הוא אופרטור שבו אנו יכולים להגדיר אופרטור חדש שישמש כסיומת של ערך כלשהו שמתקבל כפרמטר לאופרטור. נרשום operator"" ולאחריו את שם האופרטור החדש ואת הפרמטר שעליו יופעל. לדוגמא, האופרטור i שמייצר מספר מרוכב עם חלק דמיוני:

Complex operator"" i(unsigned long x) {

return Complex(0, (double)x); // use it like this – 2i

{

# תבניות

## הגדרה

תבניות הן דרך לכתוב פונקציה כללית (גנרית) המתאימה לסוגי משתנים שונים, אבל מתבצעת בצורה שונה לכל סוג. דוגמה פשוטה היא פונקציה להחלפה בין שני משתנים: swap (a, b). אפשר לכתוב פונקציה כזו שתחליף בין משתנים מסוג int, ופונקציה נוספת שתחליף בין משתנים מסוג string, וכן הלאה לכל סוג משתנה. בכל המימושים המימוש יהיה זהה לחלוטין פרט לסוגי המשתנים. בעזרת תבנית של swap נוכל לכתוב את הפונקציה פעם אחת ולהשתמש בה לכל סוגי המשתנים.

בשפת C יכולנו לעשות זאת בעזרת שליחה לפונקציה מצביע כללי (void\*), אך זה בעייתי מכמה סיבות:

1. אין בדיקה שהסוגים אכן תואמים.
2. צריך להעביר לפונקציה את גודל הסוג שרוצים להחליף.
3. הביצוע פחות יעיל, כי צריך להעביר בית בית.

## הצהרה

כדי להגדיר תבנית כזו נצהיר בתחילת הפונקציה template <typename T> . הצהרה זו אומרת לקומפיילר שפונקציה זו היא בעצם תבנית, כאשר T הוא פרמטר-סוג שיתאים את עצמו עבור כל שימוש בתבנית זו. לא בהכרח חייב לכנות את הפרמטר סוג T, אלא אפשר להשתמש בכל כינוי שנרצה. מקובל אמנם להשתמש באותיות אנגליות גדולות כמו: T, K, G, E, או שם אחר המבטא את סוג הטיפוס שמצפים לקבל.

template <typename T> void swap(T& a, T& b) {

T temp = a; a = b; b = temp;

ניתן לקרוא לפונקציה זו בצורה מפורשת, לדוגמא עבור טיפוס ממשי swap<double>(d1, d2). ואפשר גם בצורה לא מפורשת swap(d1, d2), והקומפיילר יבין לבד באיזה תבנית וטיפוס מדובר.

כדי להגדיר תבנית עם מספר פרמטרי-סוג נפריד בפסיקים את ההצהרה על הסוגים בסוגריים המשולשים. כמובן שניתן גם להחזיר פרמטר סוג. בנוסף, ניתן לבצע "העמסה" על תבניות, כלומר ליצור תבניות עם שם זהה אך עם פרמטרים שונים.

template <typename T, typename K,…> T example(T& a, K& b) {}

הקומפיילר משתמש בתבניות ליצירת קוד תוך כדי קומפילציה, ולכן כל תבנית חייבת להיות מוגדרת כולה, כולל המימוש, במקום שהקומפיילר יכול לראות כאשר הוא בונה את התוכנית הראשית - כלומר בקובץ hpp. אם נגדיר בקובץ hpp כותרת של תבנית בלי מימוש, הקומפיילר לא יוכל להשתמש בתבנית זו.

## כיצד תבניות עובדות

כשמגדירים תבנית, הקומפיילר זוכר את ההגדרה אבל עדיין לא מייצר שום פונקציה. תבנית היא לא פונקציה, היא רק **מתכון או נוסחה לייצור פונקציה**. כאשר הקומפיילר מזהה בזמן קומפילציה קריאה לתבנית זו, הפונקציה נוצרת בהתאם לטיפוסים שהוכנסו בקריאה. לדוגמה, הקריאה swap(a, b) כאשר a, b מסוג int, תגרום ליצירת פונקציה swap(int& a, int& b). אותה קריאה בדיוק רק כאשר a, b מסוג double, תגרום ליצירת פונקציה אחרת שבה הפרמטרים מסוג double&. התהליך של יצירת פונקציה מתוך תבנית נקרא instantiation. לעומת זאת, אם כתבנו תבנית בקוד אך לא הוספנו קריאה לתבנית זאת, הקומפיילר לא יצור כלום.

סדר החיפוש בכל פעם שהקומפיילר נתקל בקריאה לפונקציה, הוא קודם אוסף את כל הפונקציות והתבניות שיש להם אותו שם ומספר פרמטרים. מתוך קבוצה זו הוא קודם בוחר את כל הפונקציות עם רמת ההתאמה הגבוהה ביותר, כלומר אלו שמצריכות הכי פחות המרות סוגים, כמו שתיארנו [בהעמסת פונקציות](#_העמסת_פונקציות). רק אם לא נמצאה אף פונקציה כזו הוא מחפש את התבנית הכי מתאימה ליצור ממנה פונקציה.

סדר חיפוש זה בעצם מאפשר לנו לכתוב תבנית לפונקציה כללית, ויחד איתה פונקציה ספציפית יותר הפועלת באופן שונה על סוגים שונים. הקומפיילר יבחר את הפונקציה הספציפית אם היא מתאימה, אחרת יבחר את הפונקציה הכללית יותר. לדוגמה, נוכל לבנות פונקציה swap עבור טיפוסים מספריים, שבהם אפשר לבצע החלפה בעזרת פעולות חיבור וחיסור בלי משתנה זמני (כדי לחסוך בזיכרון). ובנוסף, תבנית swap עבור טיפוסים לא מספריים שתתבצע עם משתנה זמני. נפרט על כך בהמשך.

## שגיאות

אם בתוך המימוש של התבנית כתבנו קוד שגוי, אזי הקומפיילר לא יזהה שגיאה מיד. רק כאשר הקומפיילר יגיע לקריאה לתבנית זו וינסה ליצור את הפונקציה המתאימה, אז יזהה את השגיאה. לכן ההפניה לשגיאה לא תהיה בתבנית עצמה אלא בקריאה לתבנית. השגיאה תהיה:

"in instantiation of function template specialization foo<types>"

מסיבה זו השגיאות בתבניות הן הרבה פעמים לא ברורות, ולכן חשוב מאוד לשים לב שהמימוש והקריאה לתבניות יהיה פשוט וברור וללא שגיאות.

בתוך המימוש של התבנית, אנו יכולים להפעיל שיטות ואופרטורים על הפרמטר-סוג. התבנית מניחה שכל פעולות אלו הן אכן חוקיות כי אין לה דרך לדעת עבור איזה סוגים יפעילו אותה. לכן חשוב לשים לב שכל שיטות ואופרטורים אלו אכן מוגדרים עבור הסוגים שנשתמש בהם בקריאה לתבנית, אחרת נקבל שגיאת קומפילציה. לדוגמה, התבנית swap(T& a, T& b) מניחה של-T יש בנאי מעתיק שבאמצעותו אפשר ליצור אובייקט חדש מסוג T, וכן מניחה שאפשר להעתיק את אובייקט a לתוך אובייקט b, כלומר ל-T יש אופרטור השמה.

בחלק מהמקרים, אם לא מוגדרים שיטות ואופרטורים אלו, אזי הקומפיילר יצור אותם עבורנו כברירת מחדל, כמו במקרה של בנאי מעתיק ואופרטור השמה. אמנם אם לא ניתן לעשות זאת, כמו לדוגמה במקרה שהם מוגדרים פרטיים, תוחזר שגיאת קומפילציה. כאמור לעיל, השגיאה לא תהיה על התבנית אלא על הקריאה לתבנית.

האופרטורים המרכזיים שבדרך כלל יש לשים לב שהם מוגדרים בסוג האובייקט עליו אנו קוראים לתבנית הם: בנאי מעתיק, ואופרטורי השמה והשוואה. כמובן לוודא שהם אכן ציבוריים.

## תבניות מחלקה

עד כאן ראינו תבניות של פונקציות. באותו אופן אפשר להגדיר תבניות של מחלקות, כלומר מחלקות שכל השדות הפנימיים או חלקן הם מסוג הפרמטר-סוג שהכנסנו לתבנית. השימוש העיקרי הוא במחלקות המייצגות מיכלים גנריים שנרצה שיאחסנו סוגי אובייקטים שונים לפי בחירתנו. ההצהרה על מחלקה כזו תהיה בדרך הבאה:

template <typename T> class Class\_name {

T x; T y;

};

נוכל להגדיר שהמחלקה תכיל כמה פרמטרי סוג, על ידי כך שנפריד בפסיקים את ההצהרה על הסוגים בסוגריים המשולשים. כמו בפונקציות, גם כאן המימוש המלא של המחלקה חייב להיות ממומש בקובץ hpp. למחלקות שהן תבניות אין קובץ cpp. זאת מפני שכל תבנית חייבת להיות מוגדרת כולה, במקום שהקומפיילר יכול לראות כאשר הוא בונה את התוכנית הראשית. אם נגדיר בקובץ hpp כותרות של המחלקה בלי מימוש, הקומפיילר לא יוכל להשתמש בתבנית זו.

לדוגמה, אם נרצה להגדיר מחלקה המייצגת מחסנית גנרית שתוכל להכיל סוגי אובייקטים שונים, נגדיר את המחסנית והשדות שלה בדרך הבאה:

template <typename T> class Stack {

struct Node {

T value;

Node\* next;

Node(const T& v, Node\* n) : value(v), next(n){}

};

Node\* first;

};

כדי ליצור אובייקט מתבנית מחלקה, צריך להגיד לקומפיילר בשלב ההצהרה איזה סוג לשים במקום T.

Stack<int> intList; // T = int

Stack<string> stringList; // T = string

## איטרטור

איטרטור הוא תבנית עיצוב שמטרתו לעבור על כל האיברים של מיכלים (Containers) באופן סדרתי. בפועל האיטרטור יהיה אובייקט מסוים שמצביע על איבר כלשהו במיכל. בפעם הראשונה שניצור אותו יצביע לאיבר הראשון במיכל, ובאמצעות אופרטורים ושיטות המוגדרות על האיטרטור, נוכל לקדם אותו כל עוד לא הגענו לאיבר האחרון במיכל. איטרטור הוא חובה עבור מיכלים שאינם פרימיטיביים, כלומר האיברים שלהם לא מסודרים באופן רציף בזיכרון, מפני שזוהי הדרך הנוחה והמקובלת ביותר לעבור באופן סדרתי על כל האיברים של המיכל.

כדי להגדיר איטרטור עבור מיכל כלשהו, ניצור מחלקה פנימית בתוך המיכל בשם iterator, שתכיל שדה שהוא מצביע לאיבר במיכל. כעת כדי שנוכל להתייחס אל אובייקט מסוג האיטרטור כמו אל פוינטר, נצטרך להגדיר בתוכו את כל האופרטורים הבאים:

* השמה - '='
* הגדלה ב-1 - '++' משני הצדדים.
* השוואה - '==', '=!'
* גישה לאיבר שעליו מצביעים - '\*', '<-'

כדי לשכלל את האיטרטור נוכל להוסיף את האופרטורים: הקטנה ב-1 - '--' משני הצדדים, וגישה למיקום ספציפי - '[]'.

בנוסף נגדיר במיכל שני שיטות שמחזירות אובייקט מסוג iterator. אחת בשם begin שתחזיר איטרטור לאיבר הראשון במיכל, והשנייה בשם end שתחזיר איטרטור לאיבר **שאחרי** האיבר האחרון. כעת נוכל ליצור איטרטור שמצביע לאיבר הראשון במיכל, וכל עוד הוא שונה מהאיטרטור המצביע לאיבר שאחרי האחרון במיכל נוכל לקדם אותם, כך שנעבור על כל האיברים במיכל.

### לולאות עם איטרטור

לדוגמה, מעבר בלולאה והדפסה של כל האיברים במחסנית באמצעות איטרטור:

for(Stack<int>::iterator i=myStack.begin(); i!=myStack.end(); i++){

cout << \*i << endl;

החל מ C++11, יש צורה קצרה יותר לכתוב את הלולאה עבור מיכלים שממשים את begin ו-end ואת האופרטורים שתיארנו לעיל. לולאות מסוג זה נקראות לולאות for each. אם נשתמש בהן ללא שהגדרנו את כל האופרטורים והשיטות הנצרכות, נקבל שגיאת קומפילציה. הקומפיילר יציע לנו כל מיני פונקציות דומות (candidates) למה שהוא חיפש בפועל. מתוכן נוכל להבין איזה אופרטור לא מימשנו.

for(int i : myStack) {

cout << i << endl;

}

### const\_iterator

מקובל מאוד להגדיר בנוסף למחלקת איטרטור פנימית גם מחלקת const\_iterator, שתשמש במקרים שלא רוצים לשנות את המיכל אלא רק לעבור על כל האיברים ולבצע פעולה כלשהי. שימוש נוסף הוא כאשר אנו מקבלים מיכל קבוע ואנו רוצים לקבל איטרציה שלו. מכיוון שהמיכל קבוע לא נוכל להפעיל עליו שיטות begin() ו-end() המחזירות איטרטור רגיל, אלא נצטרך לקרוא לשיטות המחזירות const\_iterator.

המחלקות זהות לחלוטין, אלא ש-const\_iterator באופרטורים '\*' ו-'<-' מחזיר const של האובייקט אליו מצביע. מקובל שהשיטות שמחזירות const\_iterator התחלה וסוף הן cbegin ו-cend. לא ניתן להשתמש בלולאת for each ב-const\_iterator.

### שילוב בין מבני נתונים

שימוש מתוחכם באיטרטורים הוא כאשר אנו בונים מבנה נתונים שאנו מעוניינים לשלב בו מבני נתונים שונים, אלא שאנו רוצים שזה יתאים לכל מבנה נתונים אפשרי. לדוגמה, נרצה שיהיה בנאי גנרי שיקבל כל סוג של מבנה נתונים ויוכל להכניס את כל האיברים במבנה נתונים זה אל המבנה נתונים שאנו בונים. לשם כך נגדיר בנאי או שיטת insert גנרית, שאליה נוכל להכניס איטרטור של כל מבנה נתונים שנרצה, ואף פוינטר רגיל של C, ואז להכניס בלולאה את כל האיברים במבנה נתונים אחד אחר השני.

Template <typename IT> Stack(const IT\* begin, const IT\* end) {

for(; begin!=end; begin++)

push(\*begin);

};

יש כאן חיסכון גדול בכתיבת קוד. נניח שיש לנו m מבני נתונים שונים. בלי תבניות ואיטרטורים, היינו צריכים m^2 בנאים, כל מבנה נתונים יגדיר בנאי עבור כל מבנה נתונים אחר. עם תבניות ואיטרטורים, אנחנו צריכים רק בנאי ואיטרטור בכל מבנה נתונים.

## קביעת סוג ברירת מחדל

עד כה ראינו תבניות מחלקות שבהצהרה שלהן הוגדרו רק פרמטרי סוג כלליים <typename T>. אמנם אפשר גם להגדיר תבניות שיכולות לקבל פרמטרים שהם סוגים ספציפיים. לדוגמה, אפשר ליצור מחלקה המייצגת מערך שהגודל שלו חייב להיות ידוע כבר בזמן קומפילציה.

template <typename T, **int size**> class array {

T values[size];

מה מיוחד במערך כזה ביחס לווקטור ומערך רגיל? ההבדל הוא שמצד אחד, בניגוד לווקטור, כל האיברים במערך שמורים על המחסנית ולא בערימה. ומצד שני, בניגוד למערך רגיל, נוכל להוסיף למבנה זה שיטות ואופרטורים כך שיהיה לנו יותר נוח להשתמש בו, כמו אופרטור [] שבודק חריגה, הדפסה, וכו'. בנוסף, העובדה שגודל המערך ידוע בזמן קומפילציה מאפשרת לקומפיילר לבצע אופטימיזציות מעניינות. למשל, אם יוצרים מערך בגודל size=3, ואחת השיטות שלו היא לולאה מ-0 עד size, ייתכן שהקומפיילר בכלל לא ייצור לולאה אלא שלוש קריאות לגוף הלולאה, ובכך יחסוך את משתנה הלולאה ואת הקפיצות קדימה.

כמו שאנו מכירים מפונקציות, אפשר להעביר ערכי ברירת מחדל לפרמטרי-סוג וגם לפרמטרים אחרים. בהמשך לדוגמא הקודמת, נוכל להגדיר את המחלקה כך:

template <typename T=char, int size=1024> class array {

T values[size];

במקרה כזה המחלקה array<char, 2014> והמחלקה array<char> מייצגות בעצם את אותה המחלקה, ולכן ניתן לשים אובייקט מסוג אחד בסוג השני. אך כל שינוי קטן, כמו array<char, 1025> מהווה מחלקה אחרת לחלוטין.

בנוסף, אפשר גם להגדיר תבנית עבור פרמטר סוג T, וכבר בהצהרה להגדיר פרמטר נוסף מסוג T שיהיה ערך הברירת מחדל עבור T. כך נוכל לחייב את המשתמש לתת ערך ברירת מחדל למשתנה שהכניס. לדוגמה, עבור המחלקה המייצגת מערך, נוכל להשתמש בערך הברירת מחדל כדי לאתחל את כל האיברים במערך לערך זה.

Template <typename T, int size, T defValue> class array {

T values[size];

array() {

for (int i=0; i<n; i++)

values[i] = defValue;

}

};

הקריאה למחלקה זו תהיה בדרך הבאה: array<int,10,0> arr.

## תבניות מומחיות (Specialization)

כמו שהסברנו [לעיל](#_כיצד_תבניות_עובדות), בתגובה לקריאה לתבנית הקומפיילר קודם מחפש תבנית הכי מתאימה ורק אם לא מצא מחפש תבנית כללית. סדר חיפוש זה מאפשר לנו לכתוב תבנית למקרה כללי, ויחד איתה תבנית למקרה ספציפי יותר אותו נרצה דווקא לממש בצורה שונה מהמקרה הכללי. ההגדרה של תבנית מחלקה למקרה ספציפי תהיה בדרך הבאה (וכן באופן די דומה עבור תבנית פונקציה):

template <class T> class Test {…}; // General template

template <> class Test<int> {…}; // Specialized template for int

Test<char> b; // General

Test<int> a; // Specialized

נביא מספר דוגמאות שימושיות לתבניות מומחיות:

* לעיל ראינו תבנית עבור פונקציה swap גנרית המבצעת החלפה בעזרת בנאי מעתיק ואופרטור השמה. הפונקציה הזאת עובדת בכל המקרים, אמנם במקרים מסוימים תהיה לא יעילה. לדוגמה, במקרה שבו רוצים להחליף בין שני וקטורים, תתבצע העתקה העמוקה שלוש פעמים, אך היה אפשר להסתפק בהעתקה שטחית של הפוינטרים והמספרים בלבד. לכן נרצה להגדיר תבנית מומחית של swap עבור וקטורים.

template <typename T> void swap(T& a, T& b) {…}

template <typename T> void swap(vector<T>& v1, vector<T>& v2) {…}

* תבנית עבור פונקציה sort גנרית המקבלת מערכים מכל סוג שהוא וממיינת אותם. בדרך כלל נשתמש באלגוריתם quick sort מפני שהמערך יכול להכיל ערכים רבים. אמנם עבור מערך מטיפוס char, היכול להכיל לכל היותר 256 ערכים שונים, יהיה יותר יעיל להשתמש באלגוריתם counting sort. לכן עבור טיפוס char נגדיר תבנית מומחית של sort.

template<typename T> void sort(T arr[], int size) {…}

template <> void sort<char>(char arr[], int size) {…}

* תבנית מחלקה עבור וקטור יכולה להכיל כל סוג אובייקט שנרצה. אמנם עבור וקטור המכיל אובייקטים בוליאניים, אם נחזיק לכל איבר משתנה מסוג bool, התופס בית שלם, זה יהיה מאוד בזבזני. שכן היינו מעדיפים להכניס בבית אחד כזה 8 משתנים בוליאניים. לשם כך נרצה להגדיר מחלקת וקטור מומחית למשתנים בוליאניים (ראה תיקייה 9 את המימוש).

template <typename T> class vector {…};

template <>class vector<bool> {…};

## type\_traits

בהמשך לסעיף הקודם, במקרים מסוימים בהגדרת תבנית מחלקה או פונקציה המקבלת פרמטר סוג כללי, נרצה שהיא תפעל רק על משתנים מסוג מסוים. וכן יכול להיות ההיפך, שנרצה שתפעל על כל סוגי המשתנים מלבד סוג משתנים מסוים. לדוגמה, פונקציה המקבלת מערך של אובייקטים ומבצעת עליהם מיון באמצעות אופרטור >, בוודאי תיצור בעיות עבור מערך המכיל מצביעים. שכן אופרטור זה מתייחס לכתובות בזיכרון של המצביעים ולא על הערכים אליהם מצביעים.

אפשרות אחת לפתרון הבעיה היא להגדיר תבנית מומחית עבור כל סוג מקבוצה זו. אמנם ברור שבחלק מהמקרים זה יהיה מאוד מייגע לכתוב המון תבניות מומחיות כאלו. לשם כך יש בספרייה הסטנדרטית של cpp ספרייה בשם <type\_traits> שמכילה המון תבניות מחלקה, וכל אחת מיועדת לקבוצה שונה של סוגי משתנים. כל תבנית מחלקה כזו מחזיקה משתנה בוליאני סטטי ציבורי בשם value, שיקבל true אם הסוג שהכנסנו לתבנית אכן בקבוצה שהיא מייצגת, אחרת false. נביא כמה דוגמאות:

is\_pointer<int>::value // false

is\_pointer<int\*>::value // true

is\_integral<float>::value // false

is\_integral<int>::value // true

אפשר לראות את כל סוגי התבניות בספרייה [כאן](https://en.cppreference.com/w/cpp/header/type_traits). באמצעות תבניות אלו, נוכל בתוך תבנית מחלקה או פונקציה שבנינו המקבלת פרמטר סוג T, לוודא באמצעות static\_assert ש-T הוא אכן סוג שאנו מצפים לקבל, ואחרת תיזרק שגיאה. נשתמש ב-static\_assert ולא ב-assert רגיל, מפני שהבדיקה של המחלקות מספריית type\_traits מתבצעת בזמן קומפילציה בלבד ולא בזמן ריצה.

Template <typename T> T div(T a, T b) {

static\_assert(is\_integral<T>::value, "Can only be numeric type");

}

## decltype

עוד כלי שימושי בשימוש בתבניות הוא decltype. הרבה פעמים במהלך כתיבת תבניות אנו משתמשים בתבניות אחרות שאנו לא יכולים לדעת מהו הסוג T שיכניסו אליהן. decltype זוהי שיטה שמקבלת אובייקט ומחזירה את הסוג של אותו אובייקט. כך אנו יכולים לבנות אובייקט מסוג שאנו לא יודעים מהו, אך כן יש לנו גישה לאובייקט כלשהו מסוג זה.

typename T::iterator inner\_itr;

decltype(\*(inner\_itr)) elem;

## תכנות-על (Meta-Programming)

מנגנון יצירת מקרים פרטיים של תבניות יכול לשמש אותנו לכתיבת תוכניות שלמות שרצות בזמן הקומפילציה. אפשר אפילו להשתמש ברקורסיה!

במצגת יש דוגמה לתבנית רקורסיבית פשוטה: התבנית Pow3 מקבלת מספר שלם ומעלה אותו בחזקת 3. הרקורסיה מתבצעת בתבנית הכללית - היא מכפילה ב-3 וקוראת לתבנית עם הפרמטר פחות 1. תנאי העצירה מתבצע ע"י יצירת מקרה פרטי שבו הפרמטר שווה 0.

דוגמה קצת יותר שימושית יש בתיקיה 5 - חישוב נומרי של הנגזרת ה-n של פונקציה כללית כלשהי.

## Proxy

פרוקסי היא תבנית עיצוב שמטרתה לקשר בין שני מחלקות שבאופן רגיל לא יכולות לתקשר ישירות. תבנית העיצוב מציעה לבנות מחלקה חדשה בפני עצמה, שתדע לקבל מידע ממחלקה אחת, לעבד את המידע, ולהעביר אותו בצורה מתאימה אל מחלקה אחרת.