



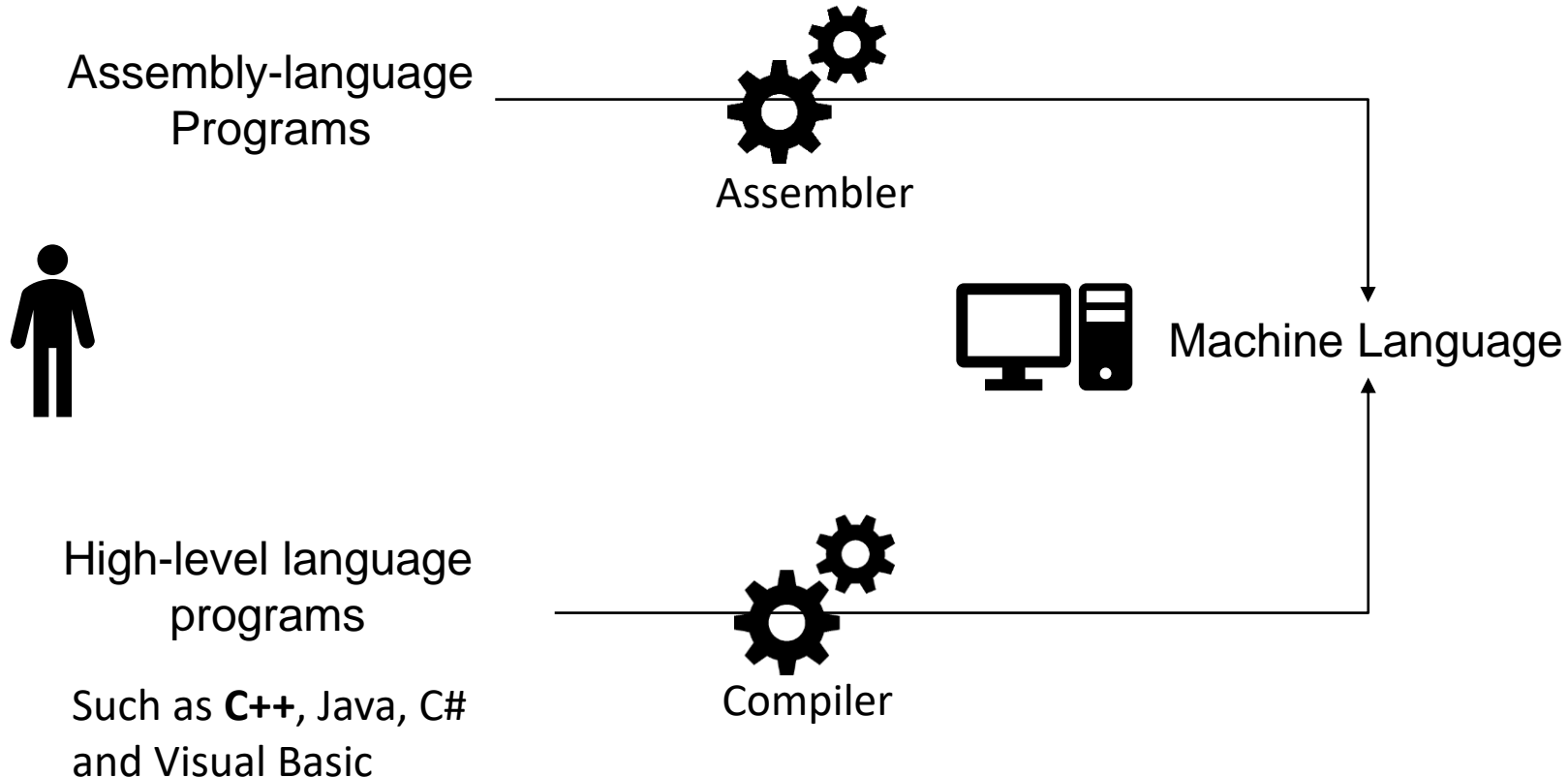
جامعة تشرين
كلية الهندسة المعلوماتية
قسم البرمجيات ونظم المعلومات

برمجة متقدمة 1 C++

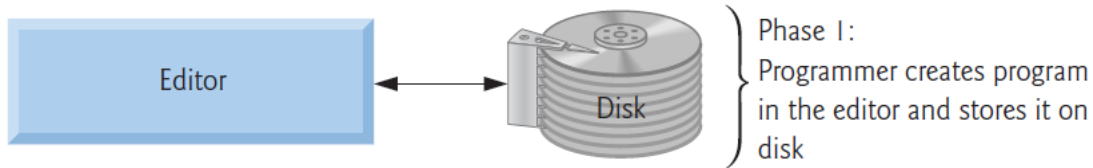
د. باسل حسن

الفصل الدراسي 2020 / 2021

- لغات الآلة Machine Languages
- لغات المجمع Assembly Languages
- لغات عالية المستوى High-Level Languages



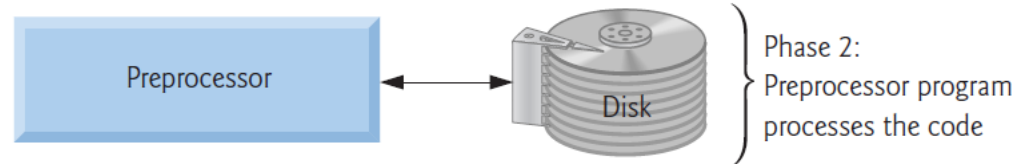
مراحل ترجمة وتنفيذ برنامج C++ - 1 / 2



Phase 1:
Programmer creates program
in the editor and stores it on
disk

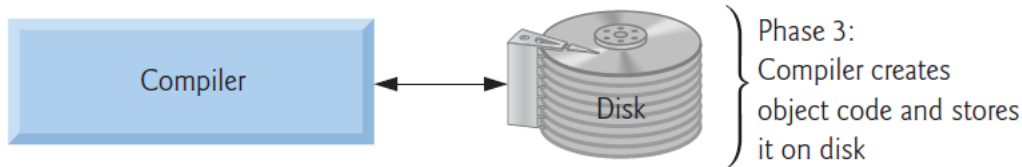
.cpp, .cxx, .cc or .C

Integrated Development Environments (IDEs)



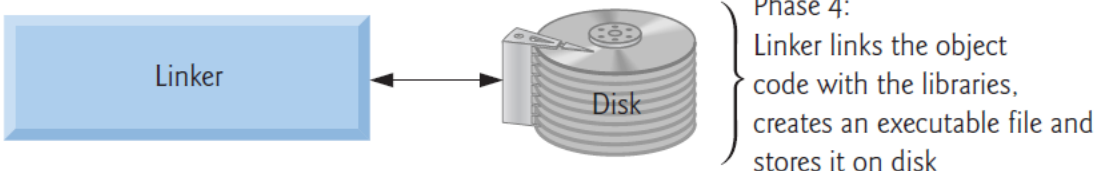
Phase 2:
Preprocessor program
processes the code

Usually include other text files to
be compiled



Phase 3:
Compiler creates
object code and stores
it on disk

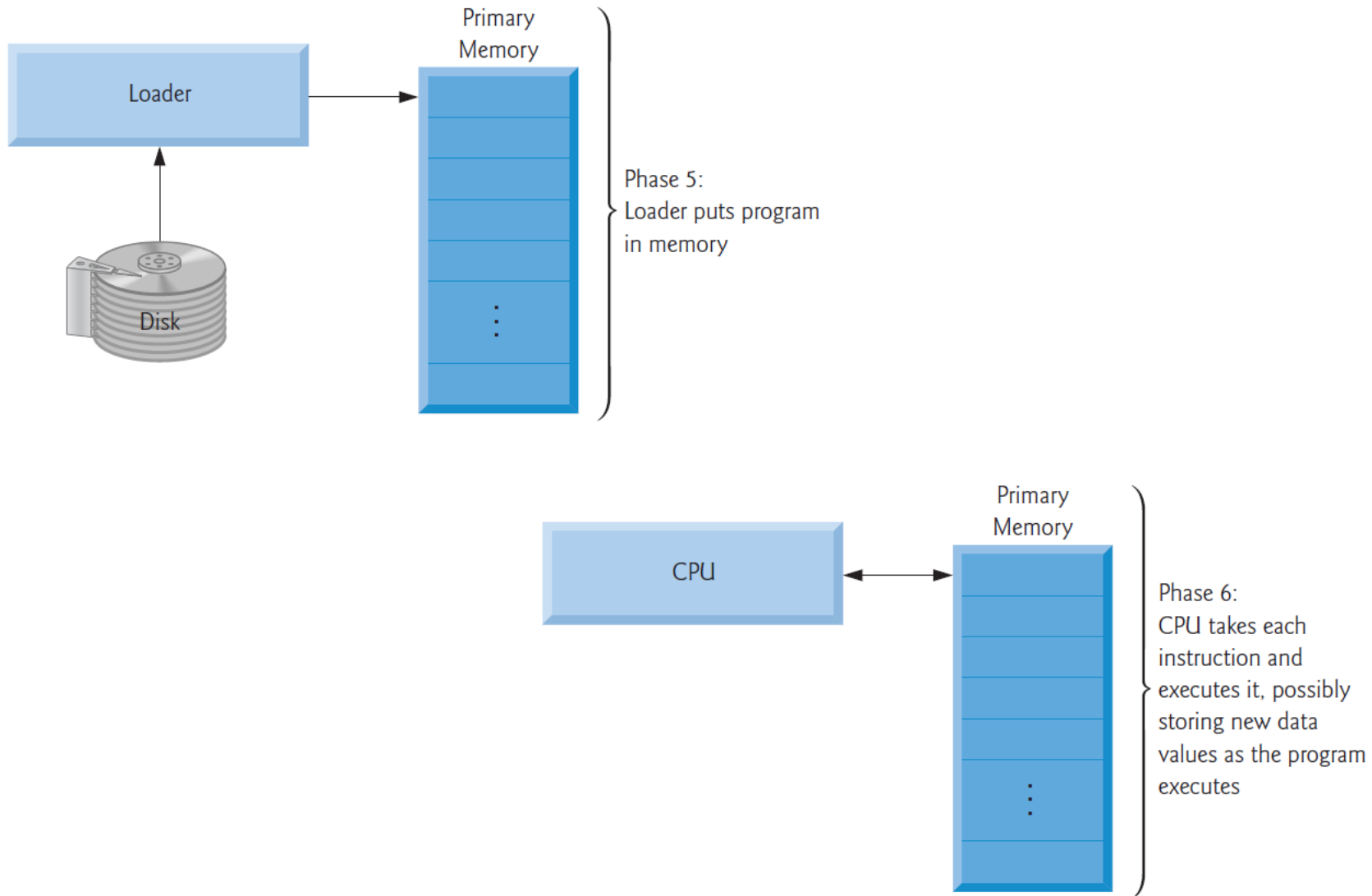
→ machine-language code



Phase 4:
Linker links the object
code with the libraries,
creates an executable file and
stores it on disk

→ executable file

مراحل ترجمة وتنفيذ برنامج C++ - 2 / 2

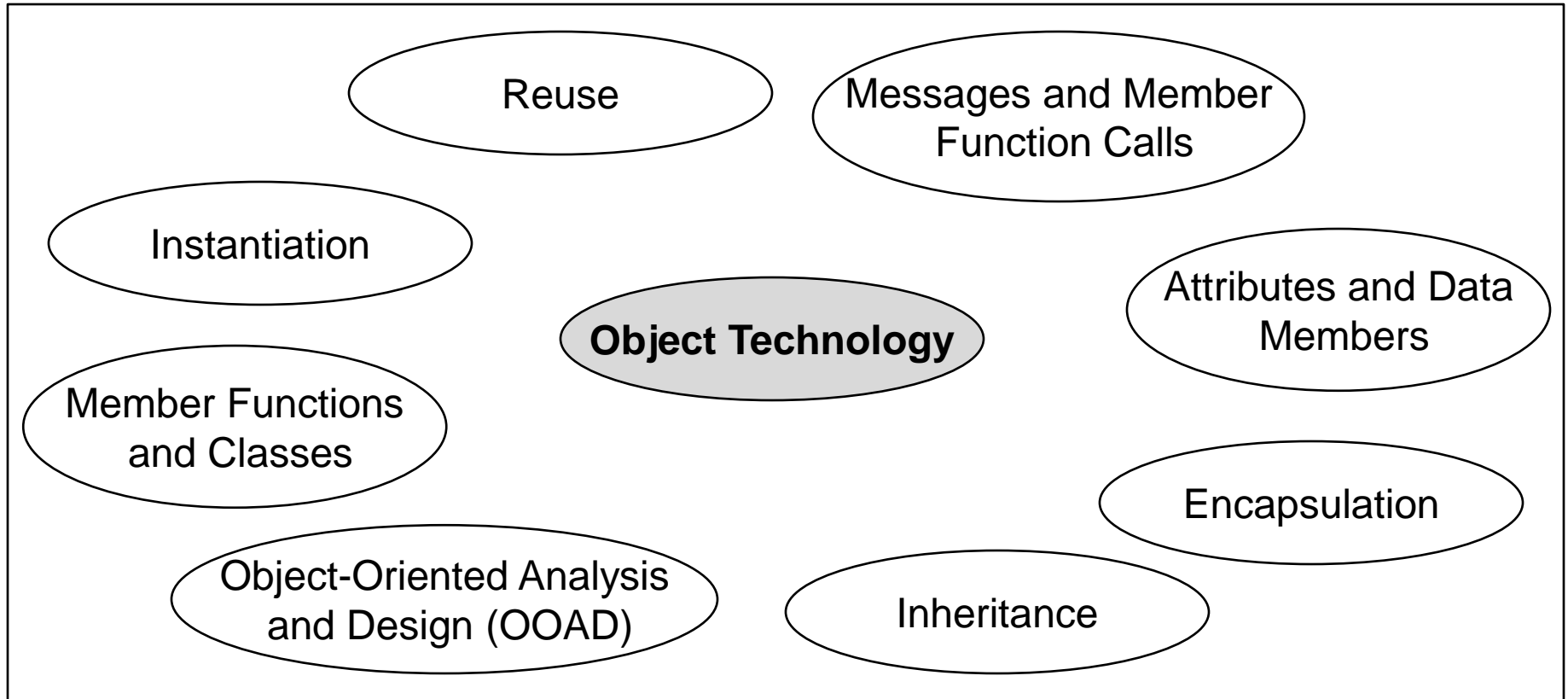


Topics RECALL

- موجّه التضمين `#include`
- التابع `main()`
- التصريح عن متحولات
 - المتحولات الثابتة (Const Variable)
- بنى التحكم (Control Structures)
 - `switch` , `if...else` , `if`
 - `while` , `for` , `do...while`
 - `continue` , `break`
- مجالات الرؤية (Scopes)
- فترة تخزين المتحولات (Variables' Storage Duration)
- التوابع (Functions)
 - طرق استدعاء التوابع
- المصفوفات, المؤشرات والمراجع

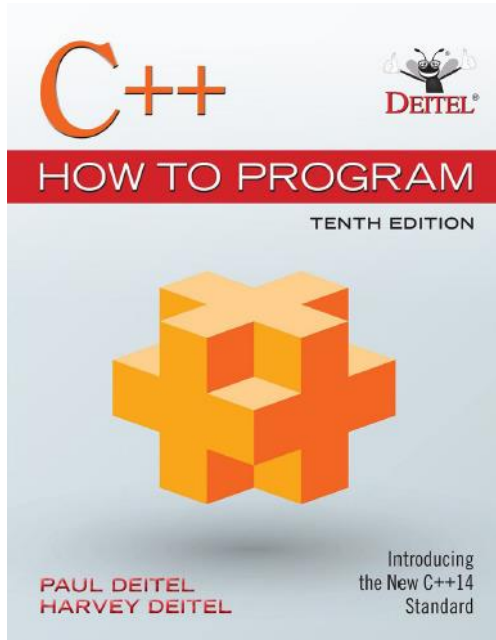
- أساسيات في لغة C++
- المصفوفات, المؤشرات والمراجع
- مقدمة إلى البرمجة غرضية التوجه
- الوراثة وتعدد الأشكال
- الواجهات
- مدخل إلى معالجة الإستثناءات
- التحميل الزائد للتوابع
- التحميل الزائد للمعاملات
- القوالب

تقنية الغرض - Object Technology



المفاهيم الأساسية للبرمجة غرضية التوجه

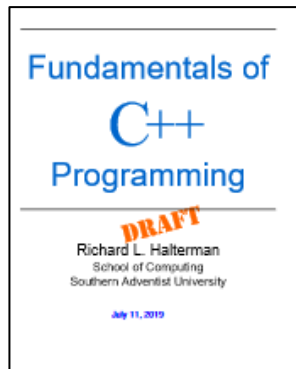
C++ is object oriented. Programming in such a language, called **object-oriented programming (OOP)**



Paul Deitel and Harvey Deitel. 2017. C++ How to Program, 10th Edition. Pearson Education.

ISBN-13: 978-0-13-444823-7

مراجع إضافية (اختيارية):



Fundamentals of
Programming C++

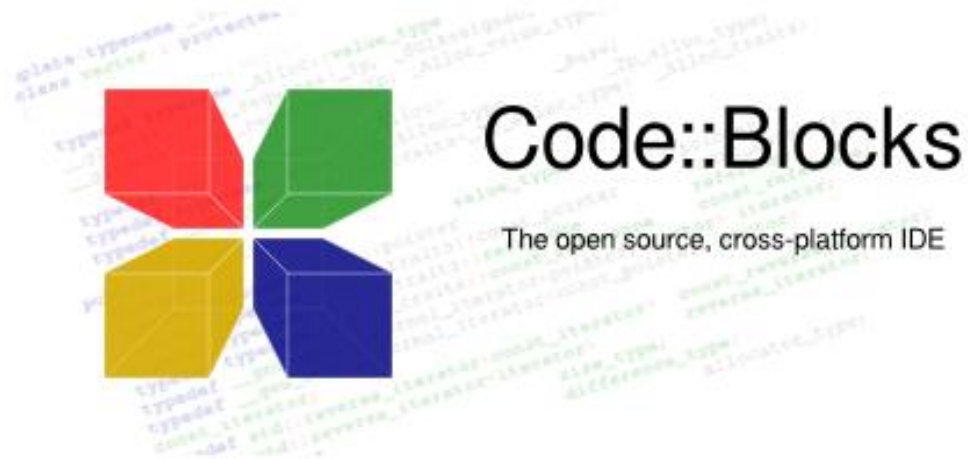
Richard L. Halterman



كيف تبرمج بلغة C++

د. صلاح الدوه جي

A Lightweight IDE for C++



Download link: <http://www.codeblocks.org/downloads>

You can also use Eclipse with MinGW GCC // Recommended

المؤشرات والمراجع

- يحتوي المتحول المؤشر (pointer variable) على عنوان ذاكرة كقيمة له
– هذا العنوان بدوره يحتوي على قيمة محددة

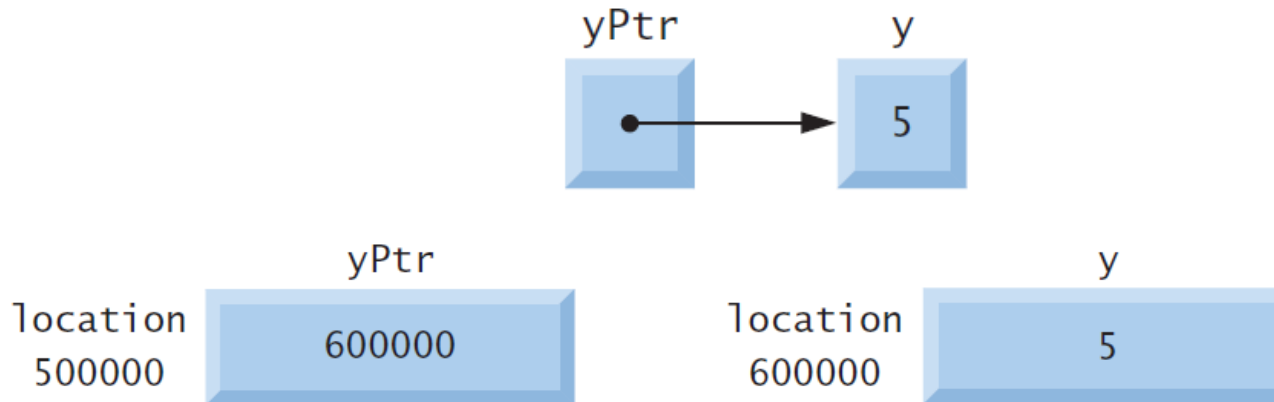
- إسناد عنوان متحول إلى المؤشر:

```
int y = 5;  
int *yPtr;  
yPtr = &y;
```

- التصريح عن مؤشر بلا قيمة أو ما يسمى بالمؤشر الصفري:

```
int *yPtr = nullptr;
```

تمثيل y و yPtr في الذاكرة



```
...  
int *yPtr;  
int y = 5;  
yPtr = &y;  
  
cout << *yPtr << endl; // 5    this is called "dereferencing"  
cout << y << endl;    // 5  
...  
*yPtr = 9;              // assign value to pointer  
cin >> *yPtr;           // input *yPtr  
...
```

```
int main() {  
    int a = 7; // assigned 7 to a  
    int *aPtr = &a; // initialize aPtr with the address of int variable a  
  
    cout << "The address of a is " << &a << "\nThe value of aPtr is " << aPtr;  
    cout << "\n\nThe value of a is " << a << "\nThe value of *aPtr is " << *aPtr  
        << endl;  
}
```

Output:

The address of a is 0x61ff08
The value of aPtr is 0x61ff08

The value of a is 7
The value of *aPtr is 7

الفرق بين المرجع والمؤشر

المؤشرات Pointers	المراجع References
يمكن التصريح عنها وتهيئتها في ما بعد	يجب تهيئتها عند التصريح عنها مباشرة
يمكن أن يعاد إسنادها	لا يمكن أن يعاد إسنادها
يمكن إسناد NULL إلى مؤشر	لا يمكن إسناد NULL إلى مرجع
يمكن تعريف مؤشر إلى مؤشر	لا يمكن تعريف مرجع إلى مرجع

المؤشرات والمراجع يُتبع...

Void Pointer

Void Pointer

المؤشر إلى النمط void هو مؤشر ليس له نوع معطيات مرتبط به

يمكن لهذا المؤشر أن يحمل عنوان من أي نوع

لا يمكن الحصول على القيمة التي يؤشر إليها بشكل مباشر. أي لا يمكن كتابة *p وإنما يجب استخدام القسر (cast)

```
int main() {  
    int a = 8;  
    void *p;  
  
    p = &a;  
  
    cout << *(int*)p << endl;  
}
```

Output:
8

- مؤشر غير ثابت يشير إلى قيمة ثابتة Nonconstant Pointer to Constant Data

```
const int *countPtr;
```

- مؤشر ثابت يشير إلى قيمة غير ثابتة Constant Pointer to Nonconstant Data

```
int *const countPtr;
```

- مؤشر ثابت يشير إلى قيمة ثابتة Constant Pointer to Constant Data

```
const int *const countPtr;
```

- تشكل المصفوفات مجموعة متتابعة من مواقع الذاكرة لها نفس النمط, هذه المجموعة لها حجم ثابت

- التصريح عن مصفوفة مدمجة:

```
type arrayName[ arraySize ];
```

- حجم المصفوفة يجب أن يكون عدد صحيح ثابت أكبر من الصفر

- بالتصريح التالي يحجز المترجم 12 عنصر للمصفوفة c:

```
int c[ 12 ]; // c is a built-in array of 12 integers
```

- يمكن تهيئة عناصر مصفوفة باستخدام قائمة التهيئة كمايلي:

```
int c[ 5 ] = { 50, 20, 30}; // results in [50, 20, 30, 0, 0]
```

```
int c[ ] = { 50, 20, 30}; // results in [50, 20, 30]
```

```
int c[ 5 ] = {}; // results in [0, 0, 0, 0, 0]
```

- يمكن استخدام المؤشرات لتنفيذ أي عملية على المصفوفات!

```
// create 5-element int array b; b is a const pointer
int b[ 5 ] = { 50, 20, 30, 10, 40 };
int *bPtr; // create int pointer bPtr, which isn't a const pointer

bPtr = b; // assign address of built-in array b to bPtr
bPtr = &b[ 0 ]; // also assigns address of built-in array b to bPtr
```

*(bPtr + 3) refers to b[3]

bPtr +3 refers to &b[3]

bPtr[1] refers to b[1]

العمليات الحسابية على المؤشرات - pointer arithmetic

- يمكن تطبيق العمليات الحسابية على المؤشرات فقط في حال هذه المؤشرات تشير إلى عناصر مصفوفة. السبب؟
- يمكن تطبيق المعاملات التالية ++, --, -=, += على المؤشرات
- يمكن طرح مؤشر من آخر فقط عندما يكون كلا المؤشرين يشير إلى نفس النمط
- يمكن إسناد مؤشر إلى آخر, عندما يكون كلا المؤشرين يشير إلى نفس النمط
- يمكن مقارنة المؤشرات ==, !=, <=, >=, <, >

الوصول إلى عناصر مصفوفة (1/2)

```
int main() {
    int b[] = { 10, 20, 30, 40 }; // create 4-element built-in array b
    int *bPtr = b; // set bPtr to point to built-in array b

    cout << "Array b displayed with:\nArray subscript notation\n";
    for (size_t i = 0; i < 4; ++i)
        cout << "b[" << i << "] = " << b[i] << '\n';

    cout << "Offset notation where the pointer is the array name\n";
    for (size_t offset1 = 0; offset1 < 4; ++offset1)
        cout << "*(b + " << offset1 << ") = " << *(b + offset1) << '\n';

    cout << "Pointer subscript notation\n";
    for (size_t j = 0; j < 4; ++j)
        cout << "bPtr[" << j << "] = " << bPtr[j] << '\n';

    cout << "Pointer/offset notation\n";
    for (size_t offset2 = 0; offset2 < 4; ++offset2)
        cout << "*(bPtr + " << offset2 << ") = " << *(bPtr + offset2) << '\n';
}
```

Output: See next slide

الوصول إلى عناصر مصفوفة (2/2) – The Output

Output:

Array b displayed with:

Array subscript notation

b[0] = 10

b[1] = 20

b[2] = 30

b[3] = 40

Offset notation where the pointer is the array name

*(b + 0) = 10

*(b + 1) = 20

*(b + 2) = 30

*(b + 3) = 40

Pointer subscript notation

bPtr[0] = 10

bPtr[1] = 20

bPtr[2] = 30

bPtr[3] = 40

Pointer/offset notation

*(bPtr + 0) = 10

*(bPtr + 1) = 20

*(bPtr + 2) = 30

*(bPtr + 3) = 40

- المعامل sizeof هو معامل أحادي يستخدم لحساب حجم وسيطه

```
cout << sizeof(char)<<endl;  
cout << sizeof(int)<<endl;  
cout << sizeof(float)<<endl;  
cout << sizeof(double)<<endl;
```

Output:

1
4
4
8

- لمعرفة عدد العناصر في المصفوفة:

```
double b[] = { 50, 20, 30, 10, 40 };  
cout << "the number of array's elements " << sizeof(b) / sizeof (b[0]);
```

- قيمة اسم المصفوفة يحوّل ضمناً إلى عنوان العنصر الأول في المصفوفة

arrayName implicitly converted to **&arrayName[0]**

- المترجم لا يفرّق بين تابع يستقبل مؤشر وتابع يستقبل مصفوفة built-in

```
//or int sumElements (const int *values, const size_t numberOfElements){  
  
int sumElements (const int values[], const size_t numberOfElements) {  
    int sum = 0;  
    for (size_t i = 0; i < numberOfElements; ++i) {  
        sum += values[i];  
    }  
    return sum;  
}  
  
int main() {  
    int b[5] = { 2, 4, 6, 8, 10 };  
  
    cout << sumElements(&b[0], 5) << endl;  
    //or cout << sumElements(b, 5) << endl;  
}
```


فترة التخزين (storage duration)

- تساعد فترة التخزين في تحديد مدة وجود معرّف (identifier) في الذاكرة
- كل المعرّفات في برنامج تملك واحد من فترات التخزين التالية:
 - automatic
 - static
 - dynamic
 - thread

فترة التخزين الأوتوماتيكي (automatic storage duration)

- **تعريف:** في فترة التخزين الأوتوماتيكي, يتم تخصيص الذاكرة للمتحول في بداية مجال الرؤية وتحريرها في نهاية مجال الرؤية
- المتحولات بفترة تخزين أوتوماتيكي تتضمن:
 - المتحولات المحلية المصرح عنها في التوابع
 - وسطاء التوابع
- التخزين الأوتوماتيكي هو مثال لمبدأ الحد الأدنى من السماحيات
(*principle of least privilege*)

فترة التخزين الساكنة (static storage duration)

- **تعريف:** في فترة التخزين الساكنة, يتم تخصيص الذاكرة للمتحول في بداية البرنامج وتحريرها في نهاية البرنامج
- يوجد نوعين من المتحولات بفترة تخزين ساكنة:
 - global variable (متحولات تم التصريح عنها خارج أي تابع أو صف)
 - local variable تم التصريح عنها بـ static
- يتم تهيئة المتحولات من النوع static مرة واحدة **فقط** عند التصريح عنها

```
static int count = 0;    // static variable
```
- إذا لم تتم تهيئة المتحولات الساكنة, تأخذ قيمة افتراضية 0 للأنماط الرقمية و الأنماط المنطقية

فترة التخزين الساكنة (static storage duration) - An Example

```
#include <iostream>
#include <iomanip>
using namespace std;

void f ()
{
    static int count = 0;    // static variable
    int i = 0;              // automatic variable
    cout << "\n" << i++ << setw(15) << count++;
}

int main()
{
    cout << "automatic" << setw(10) << "static";
    for (int ndx=0; ndx<10; ++ndx)
        f();
}
```

Output:

automatic	static
0	0
0	1
0	2
0	3
0	4
0	5
0	6
0	7
0	8
0	9

فترة التخزين الديناميكي (dynamic storage duration)

- يقوم المبرمج بمهمة بتخصيص (allocate) وتحرير (deallocate) الذاكرة للمتحول

– المعامل new لتخصيص الذاكرة
pointer-variable = **new** data-type;

```
1 int *p1 = new int;           // Allocate memory
2 int *p2 = new int(5);        // Allocate and initialize memory
3 int *p3 = new int[5];        // Allocate block of memory
4 int *p4 = new int[5]{};      // Allocate block of memory and initialize
                               // all to 0
```

– المعامل delete لتحرير الذاكرة

```
delete pointer-variable;      // to deallocate memory cases 1 and 2 above
delete[] pointer-variable;    // to deallocate block of memory
                               // cases 3 and 4 above

delete p1;
delete p2;
delete[] p3;
delete[] p4;
```

فترة التخزين الديناميكي (dynamic storage duration) - An Example

```
int main () {  
  
    int* p = new int;  
    *p = 5;  
    cout << "Value of p: " << *p << endl;  
  
    // deallocate memory  
    delete p;  
  
    int *q = new int[5];  
    for (int i = 0; i < 5; i++)  
        q[i] = i+1;  
  
    cout << "Value stored in block of memory: ";  
    for (int i = 0; i < 5; i++)  
        cout << q[i] << " ";  
  
    // deallocate block memory  
    delete[] q;  
  
    return 0;  
}
```

Output:

Value of p: 5

Value stored in block of memory: 1 2 3 4 5

```
int a[2][3];
```

- التصريح عن مصفوفة ثنائية البعد (2D array) تحتوي سطرين وثلاثة أعمدة

```
a[0][0] = 5;
```

```
a[0][1] = 19
```

```
a[0][2] = 3;
```

```
a[1][0] = 22
```

```
a[1][1] = -8
```

```
a[1][2] = 10
```

تهيئة عناصر المصفوفة

- يمكن تهيئة عناصر مصفوفة أثناء التصريح عنها كمايلي:
(تحديد عدد الأسطر هو أمر اختياري!)

```
int a[2][3] = { { 5, 19, 3 },  
               { 22, -8, 10 } };
```

```
int a[][3] = { { 5, 19, 3 },  
              { 22, -8, 10 } };
```

```
int a[][3] = { 5, 19, 3, 22};
```

3 columns

2 rows

a[0][0]	5	a[0][1]	19	a[0][2]	3
a[1][0]	22	a[1][1]	-8	a[1][2]	10

- يمكن الوصول إلى عنصر من المصفوفة كمايلي:

```
cout << a[m][n]; // Display element at row m, column n
```



```

const size_t rows = 2;
const size_t columns = 3;
void printArray( const int arr[][columns] );
int main() {
    int array1[rows][columns] = { {1,2,3},{4,5,6}};
    int array2[rows][columns] = { {1,2,3},{4,5}};

    cout << "Printing array1" << endl;
    printArray(array1);
    cout << "Printing array2" << endl;
    printArray(array2);
}

void printArray( const int arr[][columns] ) {
    // loop through array's rows
    for (size_t i = 0; i < rows; i++) {
        for (size_t j = 0; j < columns; j++)
            cout << setw(5) << arr[i][j];
        std::cout << '\n';
    }
}

```

Output:

Printing array1

1 2 3

4 5 6

Printing array2

1 2 3

4 5 0

المصفوفات متعددة الأبعاد والمؤشرات – 2 / 1

```
array-data-type *ptr = &arrayName[0][0];  
arrayName[i][j] = *(ptr+(i*columns + j))
```

```
int arr[3][4] = { {1,12,3},{8,5},{4,9,6,7}};
```

← row 0 →				← row 1 →				← row 2 →			
1	12	3	0	8	5	0	0	4	9	6	7
col 0	col 1	col 2	col 3	col 0	col 1	col 2	col 3	col 0	col 1	col 2	col 3

```
int *ptr = &arr[0][0];
```

```
for (size_t i = 0; i < 3; ++i){  
    for (size_t j = 0; j < 4; ++j)  
        cout << setw(5) << *(ptr+(i*4 + j)) << " ";  
    cout << endl;  
}
```

1	12	3	0
8	5	0	0
4	9	6	7

المصفوفات متعددة الأبعاد والمؤشرات – 2 / 2

```
int a[3][4] = { {1,12,3},{8,5},{4,9,6,7}};
```

- عنوان العنصر الأول في المصفوفة a هو $*a$
- محتوى العنصر الأول في المصفوفة a هو $**a$
- عنوان العنصر $a[i][j]$ هو $*a + (m*i) + j$ حيث m هو عدد أعمدة المصفوفة a
- محتوى العنصر $a[i][j]$ هو $*(*a + (m*i) + j)$ حيث m هو عدد أعمدة المصفوفة a

تمرير مصفوفة متعددة الأبعاد إلى تابع باستخدام المؤشرات

```
const size_t rows = 2;
const size_t columns = 3;
void printArray( int *arr);

int main() {
    int array1[rows][columns] = { {1,2,3},{4,5,6}};
    int array2[rows][columns] = { {1,2,3},{4,5}};

    cout <<"Printing array1" << endl;
    printArray(&array1[0][0]);
    cout << "Printing array2" << endl;
    printArray(*array2);
}

void printArray(int *arr ) {
    // loop through array's rows
    for (size_t i = 0; i < rows; i++) {
        for (size_t j = 0; j < columns; j++)
            cout << setw(5) << *(arr +(i*columns +j));
        std::cout << '\n';
    }
}
```

Output:

Printing array1

1	2	3
---	---	---

4	5	6
---	---	---

Printing array2

1	2	3
---	---	---

4	5	0
---	---	---

استخدام المعامل new لحجز مصفوفة ثنائية

- المصفوفة ثنائية البعد هي في الأساس مجموعة من المؤشرات إلى مصفوفات أحادية البعد.

a[0]	→	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1]	→	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2]	→	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3]	→	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

```
int **a = new int*[rows];  
for (int i = 0; i < rows; ++i)  
    *(a+i) = new int[columns];
```

- تخصيص الذاكرة

تمرين:

- استخدام حلقة for لتهيئة المصفوفة a بقيم مدخلة
- استخدم حلقة for لطباعة المصفوفة a
- اكتب التعليمات اللازمة لتحرير الذاكرة الخاصة بـ a

```
int main() {
    int **a = new int*[2]; // two rows
    for (int i = 0; i < 2; ++i)
        *(a + i) = new int[3]; // three columns
        //a[i] = new int[columns];

    for (int i = 0; i < 2; i++) {
        cout << "Enter the row #" << i << "\n";
        for (int j = 0; j < 3; j++)
            cin >> a[i][j];
    }

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++)
            cout << setw(5) << *(a[i]++) << " ";
        a[i] -= 3; // reset Pointer
        cout << endl;
    }

    //Free each sub-array
    for (int i = 0; i < 2; ++i) {
        delete[] a[i];
    }
    //Free the array of pointers
    delete[] a;
}
```

```
int main() {
    int **a = new int*[2]; // two rows
    for (int i = 0; i < 2; ++i)
        *(a + i) = new int[3]; // three columns
        //a[i] = new int[columns];

    for (int i = 0; i < 2; i++) {
        int *begin = a[i];
        cout << "Enter the row #" << i << "\n";
        for (int j = 0; j < 3; j++)
            cin >> *(a[i]++);
        a[i] = begin;
    }

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++)
            cout << setw(5) << *(a[i]++);
        cout << endl;
    }

    //Free each sub-array
    for (int i = 0; i < 2; ++i) {
        delete[] a[i];
    }
    //Free the array of pointers
    delete[] a;
}
```

- لتكن المصفوفة a معرفة كمايلي:

```
int main() {
int a[] = {1,2,3,4,5};
}
```

- عرّف مصفوفة من المؤشرات تأخذ عناوين عناصر المصفوفة a ثم قم بطباعة القيم التي تؤشر إليها عناصر المصفوفة الناتجة

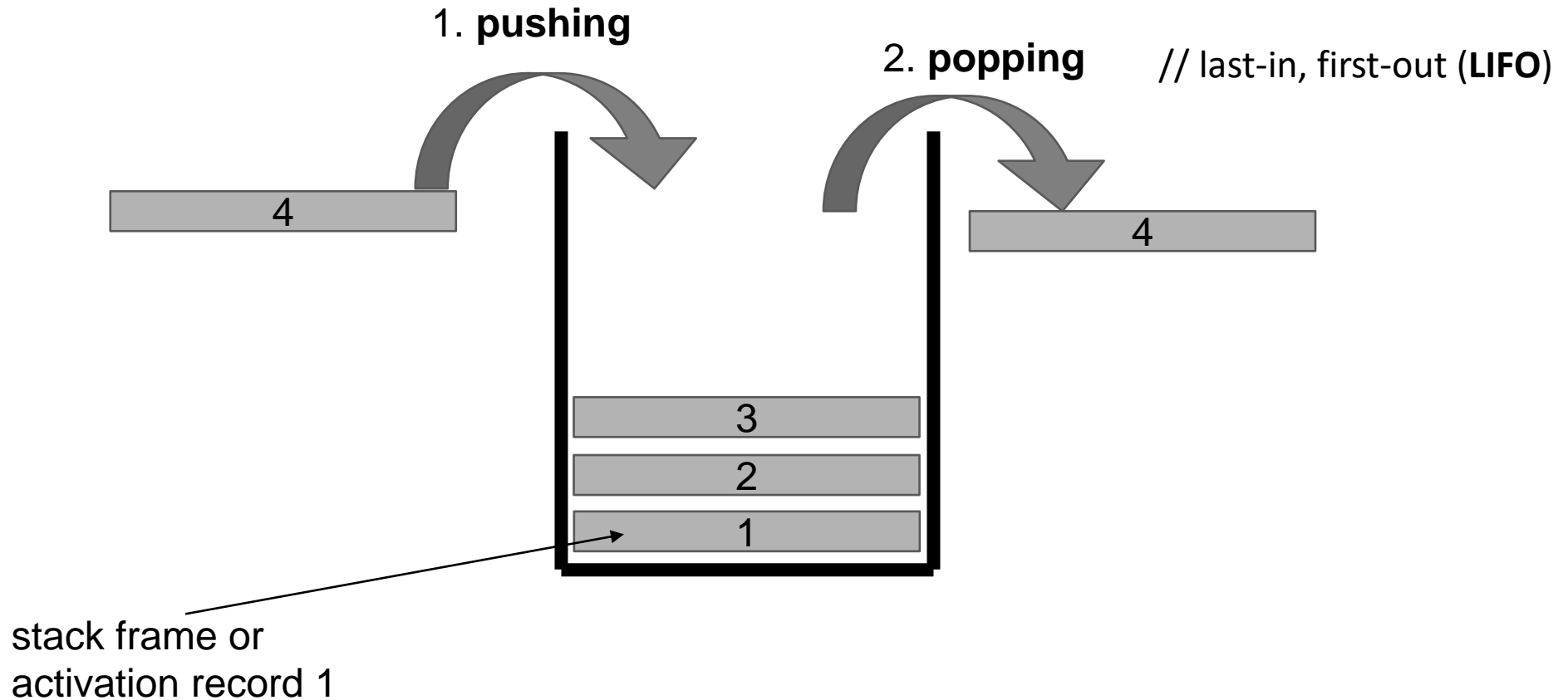
```
int main() {
int a[] = { 1, 2, 3, 4, 5 };
int *bPtr[5];

for (int i = 0; i < 5; i++) {
bPtr[i] = &a[i];
}

for (int i = 0; i < 5; i++) {
cout << *bPtr[i] << endl;
}
}
```


مكدّس استدعاء التوابع (function call stack)

- مكدّس استدعاء التوابع يدعم تقنية الاستدعاء وإعادة للتوابع
- يدعم أيضاً خلق, صيانة وتدمير المتحولات الأوتوماتيكية لكل تابع مُستدعى
- في كل مرة يقوم تابع باستدعاء تابع آخر، يتم إدخال (push) سجل في المكدّس. هذا السجل يسمى سجل التفعيل (activation record)
- يحتوي سجل التفعيل على عنوان إعادة الذي يحتاجه التابع المُستدعى لإعادة التحكم إلى التابع الذي قام بالاستدعاء
- يُعد سجل التفعيل مكاناً مثالياً لحجز الذاكرة للمتغيرات المحلية غير الساكنة (non-static local variable) للتابع الذي تم استدعاؤه
- عند إعادة من قبل التابع المُستدعى, يتم سحب (pop) سجل التفعيل لهذا التابع من المكدّس
- يجد التابع المُستدعى المعلومات اللازمة للعودة إلى التابع الذي قام بالاستدعاء في أعلى المكدّس دائماً



! في حالة حدوث المزيد من استدعاءات التتابع أكثر مما يمكن تخزين سجلات التفعيل الخاصة بها في مكدس استدعاء التتابع، يحدث خطأ يُعرف باسم تجاوز سعة المكدس (stack overflow)

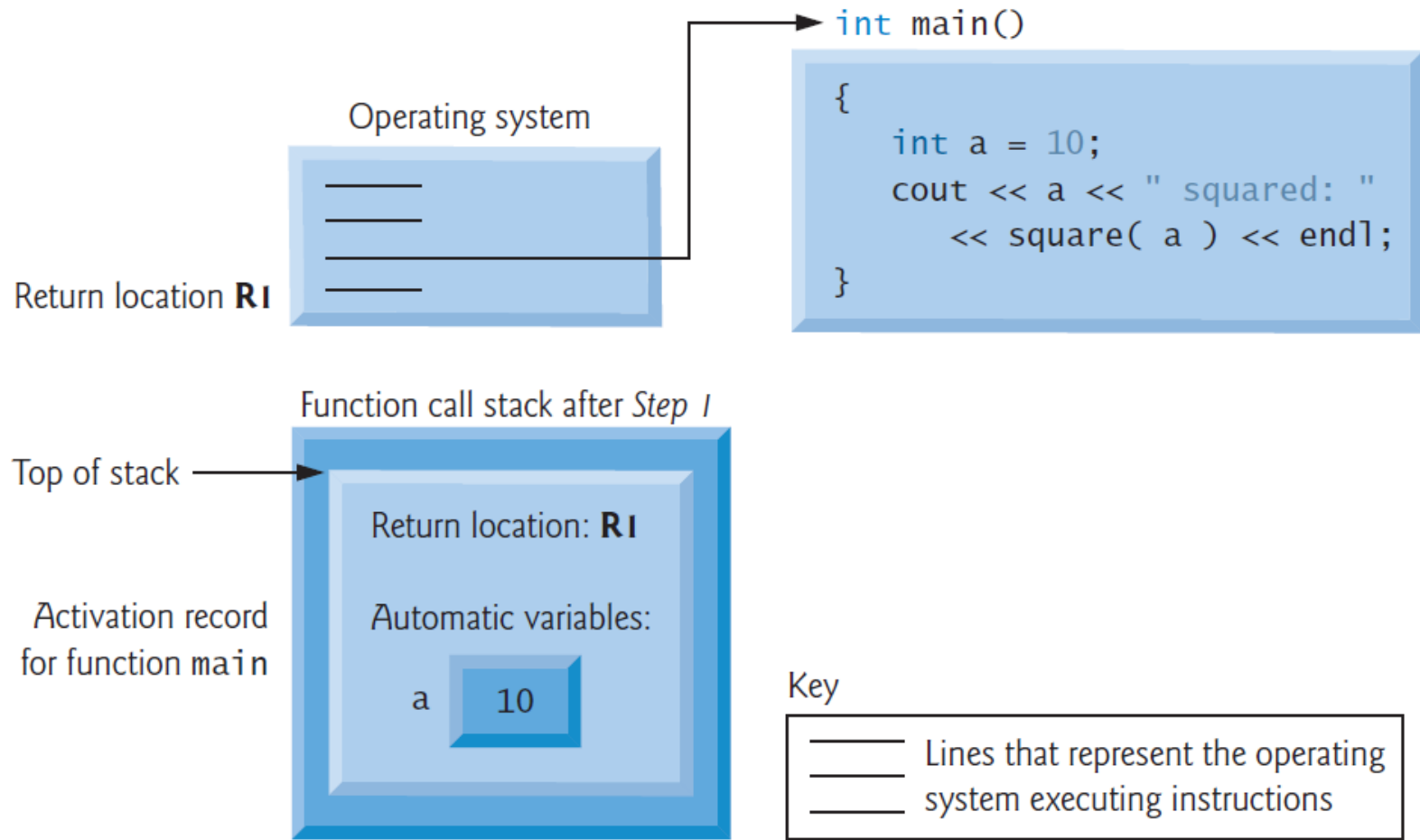
```
int square(int); // prototype for function square

int main() {
    int a = 10;
    cout << a << " squared: " << square(a) << endl;
}

// returns the square of an integer
int square(int x) { // x is a local variable
    return x * x; // calculate square and return result
}
```

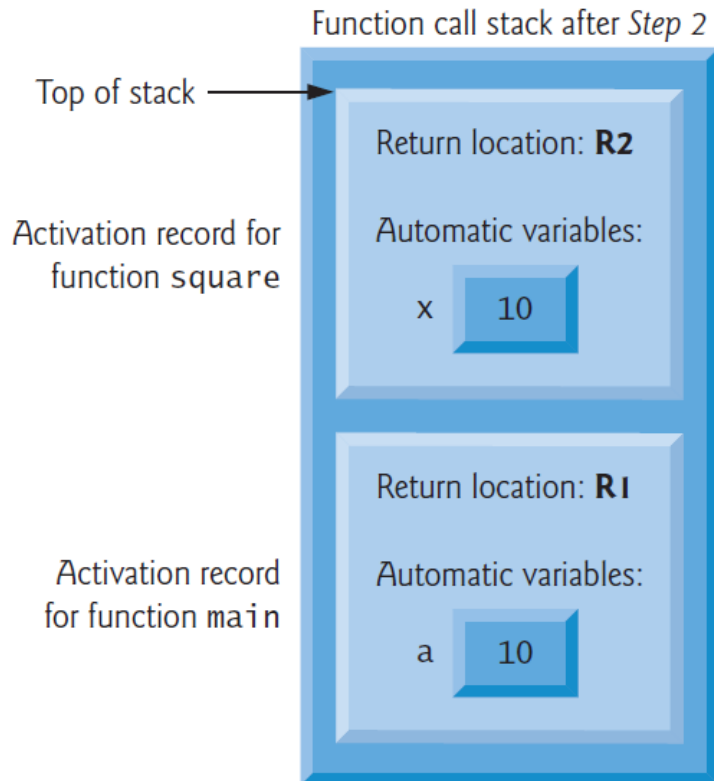
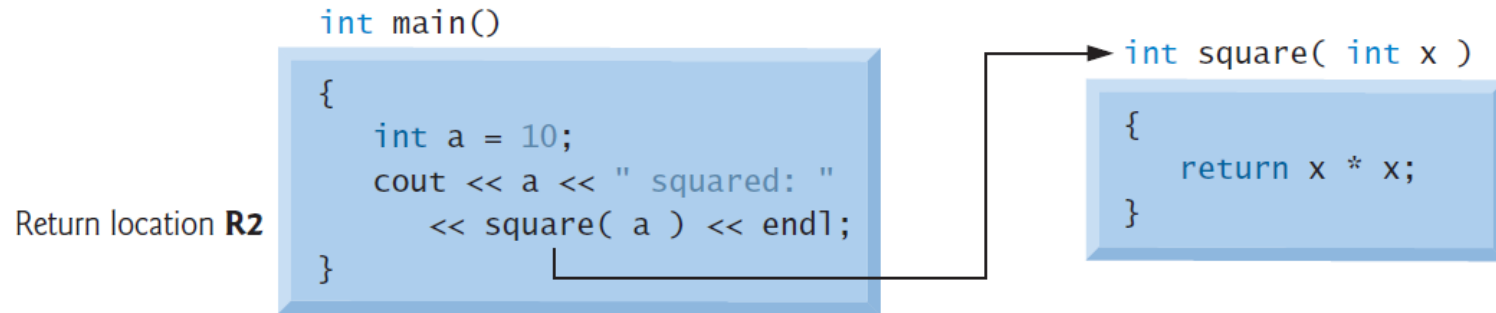
2/4 مكدّس استدعاء التوابيع (function call stack) - An Example

Step 1: Operating system invokes main to execute application



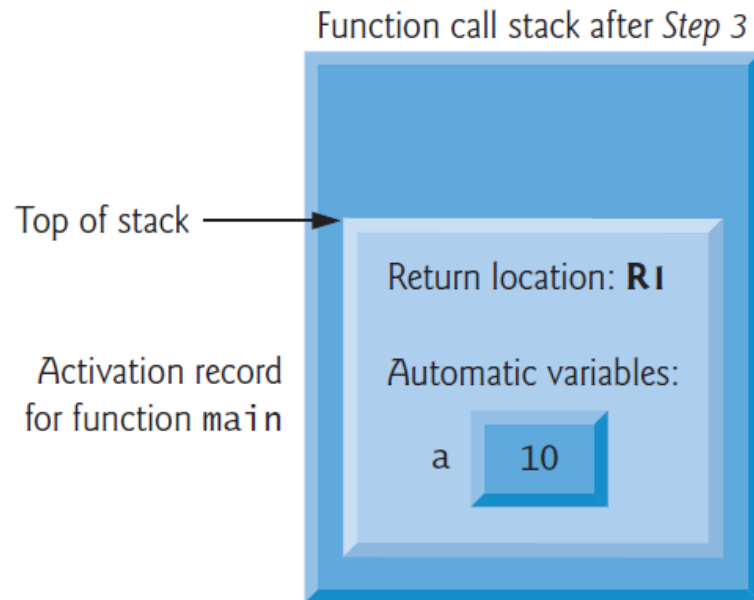
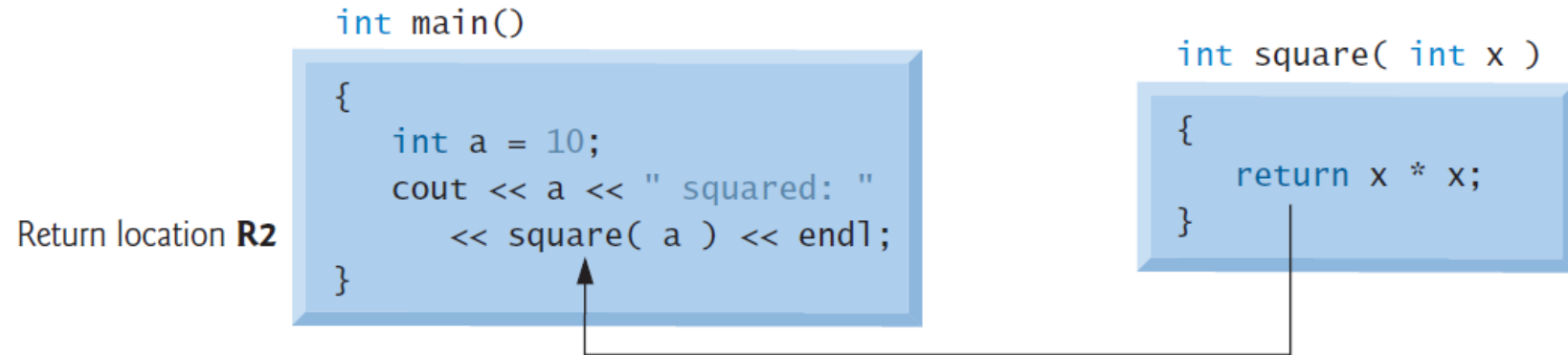
3/4 مكدّس استدعاء التوابيع (function call stack) - An Example

Step 2: main invokes function square to perform calculation



4/4 An Example - (function call stack) مكدس استدعاء التوابع

Step 3: square returns its result to main



- يوجد ثلاث طرق لتمرير الوسطاء إلى التوابع:

– التمرير بقيمة (pass-by-value)

- تمرير نسخة من قيمة المتحول إلى التابع المُستدعى
- تغيير تلك النسخة لا يؤثر على قيمة المتحول الأصلية في التابع الذي قام بالإستدعاء

– التمرير بمرجع (pass-by-reference)

- تمرير المتحول كمرجع إلى التابع المُستدعى
- التابع المُستدعى لديه إمكانية الوصول إلى متحولات التابع الذي قام بالإستدعاء بشكل مباشر, ويستطيع تغيير قيمة هذه المتحولات

– التمرير بعنوان (pass-by-address)

التمرير بعنوان (pass-by-address)

- تمرير عنوان المتحول إلى التابع المُستدعى
- التابع المُستدعى لديه إمكانية الوصول إلى متحولات التابع الذي قام بالإستدعاء بشكل مباشر, ويستطيع تغيير قيمة هذه المتحولات

```
void addOne(int *ptr) // ptr is a pointer variable
{
    *ptr = *ptr + 1;
}

int main()
{
    int value = 5;

    cout << "value = " << value << '\n';
    addOne(&value);
    cout << "value = " << value << '\n';
    return 0;
}
```


Return Value by reference

dataType& functionName(parameters);

Output:
compilation error

```
int& f1() {  
    int x = 1;  
    return x;  
}  
  
int main() {  
    f1() = 10;  
    cout << x;  
    return 0;  
}
```



Output:
10

```
int x;  
  
int& f1() {  
    return x;  
}  
  
int main() {  
    f1() = 10;  
    cout << x;  
    return 0;  
}
```



Output:
1

```
int& f1() {  
    static int x = 1;  
    return x;  
}  
  
int main() {  
    // f1() = 10;  
    cout << f1();  
    return 0;  
}
```



Output:
1

```
int* f1() {  
    int* x = new int(1);  
    return x;  
}  
  
int main() {  
    // *(f1()) = 10; //!!  
    cout << *(f1());  
    return 0;  
}
```



نمط الإعادة للتابع f1 هو مرجع للمتحول x
لذلك سيتم إسناد القيمة 10 للمتحول x

An Example - Return Value by reference

```
int& returnValue(int& x) {  
    cout << "x = " << x << " The address of x is " << &x << endl;  
  
    // Return reference  
    return x;  
}  
  
int main(){  
    int a = 20;  
    int& b = returnValue(a);  
  
    cout << "a = " << a << " The address of a is " << &a << endl;  
    cout << "b = " << b << " The address of b is " << &b << endl;  
  
    returnValue(a) = 13;  
  
    cout << "a = " << a << " The address of a is " << &a << endl;  
    return 0;  
}
```

Output:

```
x = 20 The address of x is 0x62ff08  
a = 20 The address of a is 0x62ff08  
b = 20 The address of b is 0x62ff08  
x = 20 The address of x is 0x62ff08  
a = 13 The address of a is 0x62ff08
```

++*p, *p++ and *++p

تمرين #1 أسبقية المعاملات!!

```
int main() {  
    int *p = new int[5] { 3, 5, 7, 9, 11 };  
    *p++;  
    cout << *p << endl;  
  
    cout << ++*p << endl;  
    cout << *p++ << endl;  
    cout << *++p << endl;  
    cout << *p++ << endl;  
    cout << *p++ << endl;  
    cout << *p << endl;  
}
```

- أوجد خرج البرنامج التالي:

Output:

5

6

6

9

9

11

1704128

← قيمة عشوائية!

- أوجد خرج البرنامج التالي:

Output:

4

5

5

5

5

6

7

```
int main() {  
    int *p = new int[5] { 3, 5, 7, 9, 11 };  
    (*p)++;  
    cout << *p << endl;  
  
    cout << ++(*p) << endl;  
    cout << (*p)++ << endl;  
    cout << *(++p) << endl;  
    cout << (*p)++ << endl;  
    cout << (*p)++ << endl;  
    cout << *p << endl;  
}
```

تمرين #2

- أوجد خرج البرنامج التالي:

```
int main() {
    int **a = new int*[5];
    for (int i = 0; i < 5; ++i)
        *(a + i) = new int[i+1];

    // initialize random seed:
    srand (time(0));

    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < i+1; j++)
            // generate a random number
            //between 1 and 10
            a[i][j] = rand() % 10 + 1;
    }

    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < i+1; j++)
            cout << setw(5) << a[i][j];
        cout << endl;
    }

    ...

    //Free each sub-array
    for (int i = 0; i < 5; ++i) {
        delete[] a[i];
    }
    //Free the array of pointers
    delete[] a;
}
```

```
//the address of the first element
cout << &a[0][0] << "\n";
cout << a[0] << "\n";
cout << *a << "\n";

// the value of the first element
cout << *(&a[0][0]) << "\n";
cout << *a[0] << "\n";
cout << a[0][0] << "\n";
cout << **a << "\n";

// the address of element a[3][2]
cout << &a[3][2] << "\n";
cout << a[3] + 2 << "\n";
cout << *(a+3) + 2 << "\n";

// the value of element a[3][2]

cout << *(&a[3][2]) << "\n";
cout << *(a[3] + 2) << "\n";
cout << *(* (a+3) + 2) << "\n";
cout << (*(a+3))[2] << "\n";
```

```
int main() {
    int a = 5;
    int arr[] = {3,1,5,9,10};

1    const int &a_ref = a;
2    a = 0;
3    int *b = &a;
4    a_ref = 7;
5    arr = b;
6    b = &arr[2];

    cout << "a = " << a << endl;
    cout << "*b = " << *b << endl;
    cout << "a_ref = " << a_ref << endl;
    cout << "arr[0] = " << arr[0] << endl;
    cout << "*(b+2) = " << *(b+2) << endl;
}
```

- في الأسطر المرقمة (من 1 حتى 6), يوجد تعليماتي إسناد غير صحيحتين. المطلوب:
 1. أوجد كل من هاتين التعليمتين, واذكر سبب عدم صحة كل منهما
 2. اكتب خرج البرنامج بعد حذف تعليماتي الإسناد الغير صحيحتين

Output:

```
a = 0
*b = 5
a_ref = 0
arr[0] = 3
*(b+2) = 10
```

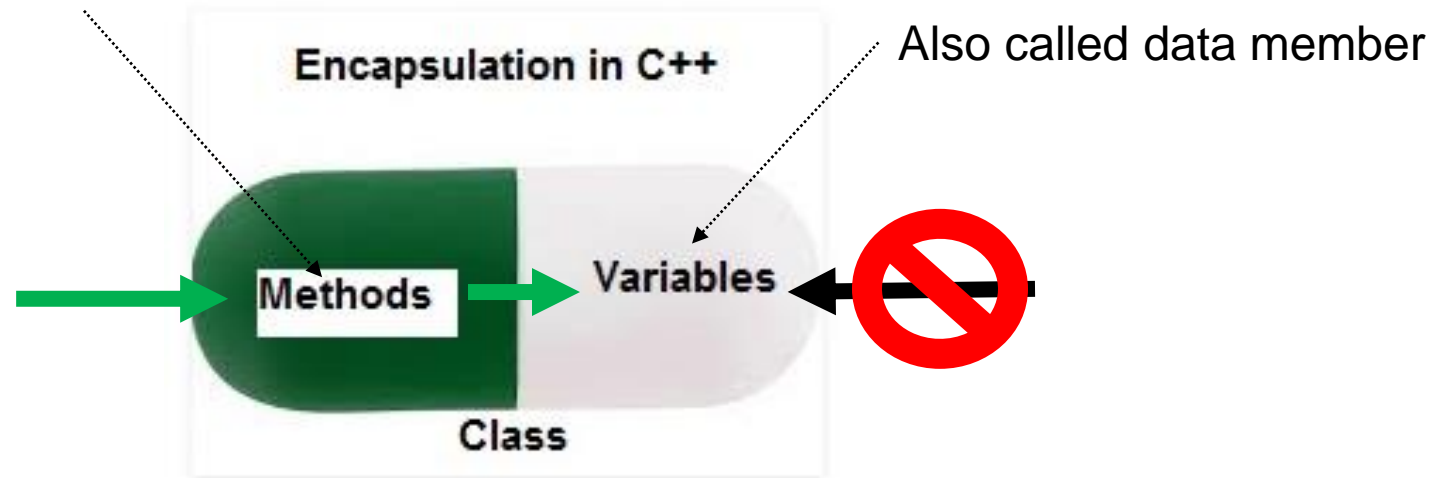
الصفوف (Classes)

- يمكن تعريف صف (نمط جديد) باستخدام الكلمة المفتاحية `class`
– يتم استخدام الصف لتغليف (encapsulate) معطيات مع طرق لمعالجتها
- المتحول من ذلك النمط يسمى حالة من صف (instance of class) أو غرض (object)
- تسمح الصفوف للغة C++ لتستخدم كلغة برمجة غرضية التوجه (Object Oriented Programming)

class vs. struct?

التغليف (Encapsulation)

Also called member functions



<https://www.geeksforgeeks.org/encapsulation-in-c/>

- تُغلف الصفوف المعطيات والتوابع الأعضاء ضمن أغراض تم إنشاؤها من هذه الصفوف
- يقوم التغليف بإخفاء البيانات (data hiding) بحيث لا يمكن الوصول لها من خارج الصف, ولكن يمكن الوصول لها عن طريق التوابع الأعضاء لذلك الصف

Class's Syntax

keyword كلمة مفتاحية

user-defined name – اسم الصف

```
class ClassName {
```

```
    Access_specifier:
```

```
    Data_members;
```

```
    Member_Functions;
```

```
};
```

private, public **or** protected

المعطيات الأعضاء
variables to be used

التوابع الأعضاء
Methods to access and manipulate
data members

```
class Account {
private:
    string name; // data member containing account holder's name
public:
    // member function that sets the account name in the object
    void setName(string accountName) {
        name = accountName; // store the account name
    }
    // member function that retrieves the account name from the object
    string getName() const {
        return name; // return name's value to this function's caller
    }
}; // end class Account

int main() {
    Account myAccount; // create Account object myAccount
    cout << "Initial account name is: " << myAccount.getName();
    cout << "\nPlease enter the account name: ";
    string theName;
    getline(cin, theName); // read a line of text
    myAccount.setName(theName);
    cout << "Name in object myAccount is: " << myAccount.getName() << endl;
}
```

Set and get member functions

```
...  
void setName(string accountName) {  
    name = accountName;  
}  
string getName() const {  
    return name;  
}  
...
```

- نسمي التابع العضو الذي يقوم بتهيئة أو تغيير قيمة لمعطيات أعضاء في صف بـ set function
- نسمي التابع العضو الذي يقوم بإعادة قيمة لمعطيات أعضاء في صف بـ get function
- يمكن إضافة الكلمة المفتاحية const إلى نهاية ترويسة تابع عضو في حال التابع لا يقوم بتغيير قيم المعطيات الأعضاء (أو في حال التابع لا يجب أن يقوم بتغيير قيم المعطيات الأعضاء)

Declaring a member function with **const** to the right of the parameter list tells the compiler, “this function should not modify the object on which it’s called—if it does, please issue a compilation error.”

محددات الوصول (Access specifiers) 1/2

- يملك الصف ثلاث محددات وصول إلى معطياته وتوابعه

- **public** تجعل المعطيات والتوابع متاحة من أي مكان ضمن البرنامج

- **private** تجعل المعطيات والتوابع متاحة فقط ضمن الصف

- **protected** تجعل المعطيات والتوابع متاحة ضمن الصف و الصفوف المشتقة منه (derived classes)
* سيتم مناقشة هذه النقطة لاحقاً ضمن موضوع الوراثة (inheritance)

محددات الوصول (Access specifiers) 2/2

```
class Account {  
private:  
    string name;  
  
public:  
    void setName(string accountName) {  
        name = accountName;  
    }  
    string getName() const {  
        return name;  
    }  
};
```

- محدد الوصول **private:** يشير إلى أنه من الممكن الوصول إلى المعطيات الأعضاء `name` من التوابع الأعضاء `setName` و `getName` فقط. هذا يعرف بإخفاء البيانات (**Data Hiding**) - من الممكن أيضاً الوصول إلى هذه المعطيات من أصدقاء الصف أيضاً (سيتم مناقشة هذه النقطة لاحقاً)
- محدد الوصول **public:** يشير إلى أنه من الممكن الوصول إلى التوابع الأعضاء `setName` و `getName` من خارج الصف `Account` حيث يتم استدعاؤها من غرض من الصف `Account`
- في حال لم تتم كتابة أي محدد وصول في بداية الصف, يكون محدد الوصول الافتراضي **private:**

إنشاء غرض من صف و استدعاء التوابع الأعضاء

```
...  
int main() {  
    Account myAccount; // create Account object myAccount  
    cout << "Initial account name is: " << myAccount.getName();  
    cout << "\nPlease enter the account name: ";  
    string theName;  
    getline(cin, theName); // read a line of text  
    myAccount.setName(theName);  
    cout << "Name in object myAccount is: " << myAccount.getName() << endl;  
}
```

- تُستدعى المشيّدات constructors لتهيئة المعطيات الأعضاء عند إنشاء غرض من صف
 - توابع بلا نمط إعادة (no return type)
 - لها نفس اسم الصف
 - يمكن تحميل المشيّد بشكل زائد!
 - إذا لم يتم تعريف مشيّد بشكل صريح, يُستخدم مشيّد ضمني افتراضي
 - لا يمكن التصريح عن مشيّد بـ `const`, لأن تهيئة غرض يعني تغييره

المشيدات (Constructors) - An Example

```
class Account {
public:
    // constructor initializes data member
    //name with parameter accountName
    Account(string accountName)
        : name{accountName} { // member-initializer list
        // empty body
    }
};
```

:name(accountName)
is also possible

```
void setName(string accountName) {
    name = accountName;
}

string getName() const {
    return name;
}

private:
    string name;
};
```

```
int main() {
    // create two Account objects
    Account account1("Jane Green");
    Account account2("John Blue");

    // display initial value of
    // name for each Account
    cout << "account1 name is: " <<
        account1.getName() << endl;
    cout << "account2 name is: " <<
        account2.getName() << endl;
}
```



```
...  
Account account1{"Jane Green"};  
Account account2("John Blue");  
...
```

- تم إنشاء غرضين (account1, account2) من الصف Account. حيث أن كل غرض يوجد في حالة خاصة به (أي كل غرض يملك معطيات تختلف عن معطيات الغرض الآخر)
- في حال المشيّد بوسطاء, يجب تمرير هذه الوسطاء ضمن أقواس أثناء التصريح عن الغرض

المشيد الافتراضي (Default Constructor)

- إذا لم يتم تعريف مشيد بشكل صريح ضمن صف, يقوم المترجم بإنشاء مشيد افتراضي بدون وسطاء
- لا يقوم المشيد الافتراضي بتهيئة المعطيات الأعضاء التي هي من الأنماط الأساسية غير الساكنة (non-static fundamental-types), ولكنه يقوم باستدعاء المشيد الافتراضي لكل من المعطيات الأعضاء التي هي أغراض من صفوف أخرى
- عند إنشاء غرض من صف بدون وضع أقواس على يمين اسم متحول الغرض, يقوم المترجم بشكل ضمني باستدعاء المشيد الافتراضي

```
class Account {  
private:  
    string name;  
public:  
  
    void setName(string accountName) {  
        name = accountName;  
    }  
  
    string getName() const {  
        return name;  
    }  
};
```

هنا سيتم استدعاء المشيد الافتراضي للصف Account
ويقوم ذلك المشيد باستدعاء المشيد الافتراضي للصف
string الذي يقوم بتهيئة الغرض name بالسلسلة
الخالية (أي "")

```
int main() {  
    Account myAccount;  
    cout << "Initial account name is: " << myAccount.getName();  
    cout << "\nPlease enter the account name: ";  
    string theName;  
    getline(cin, theName);  
    myAccount.setName(theName);  
    cout << "Name in object myAccount is: " << myAccount.getName() << endl;  
}
```

- يستدعى الهادم destructor لتحرير الذاكرة المحجوزة لغرض من صف
 - يملك الهادم نفس اسم الصف مسبق ب ~, مثال: ~SomeClass ()
 - لا يمكن تحميل الهادم بشكل زائد!
 - الهادم هو آخر تابع يتم استدعاؤه في حياة الغرض
 - إذا لم يتم تعريف هادم بشكل صريح, يُستخدم هادم ضمني افتراضي
- يملك الصف دائماً مشيّد واحد على الأقل وهادم واحد فقط

الهادم (Destructor) - An Example

```
class MyClass {  
  
public:  
    MyClass() {  
        cout << "MyClass's constructor is called" << endl;  
    }  
    ~MyClass() {  
        cout << "MyClass's destructor is called" << endl;  
    }  
};  
  
int main() {  
    MyClass myObject;  
}
```

Output:

```
MyClass's constructor is called  
MyClass's destructor is called
```

- لا حاجة إلى استدعاء الهادم بشكل صريح!

- يمكن لمشيد أن يستدعي مشيد آخر ضمن نفس الصف كمايلي: (up C++11)

```
class MyClass {
public:
    MyClass() {
        cout << "MyClass() is executed" << endl;
    }
    MyClass(int x): MyClass('a', x) {
        cout << "MyClass(int x) is executed" << endl;
    }
    MyClass(char a, int x): MyClass() {
        cout << "MyClass(char a, int x) is executed" << endl;
    }
    ~MyClass() {
        cout << "MyClass's destructor is called" << endl;
    }
};

int main() {
    MyClass myObject1;
    MyClass myObject2(5);
}
```

Output:

```
MyClass() is executed
MyClass() is executed
MyClass(char a, int x) is executed
MyClass(int x) is executed
MyClass's destructor is called
MyClass's destructor is called
```

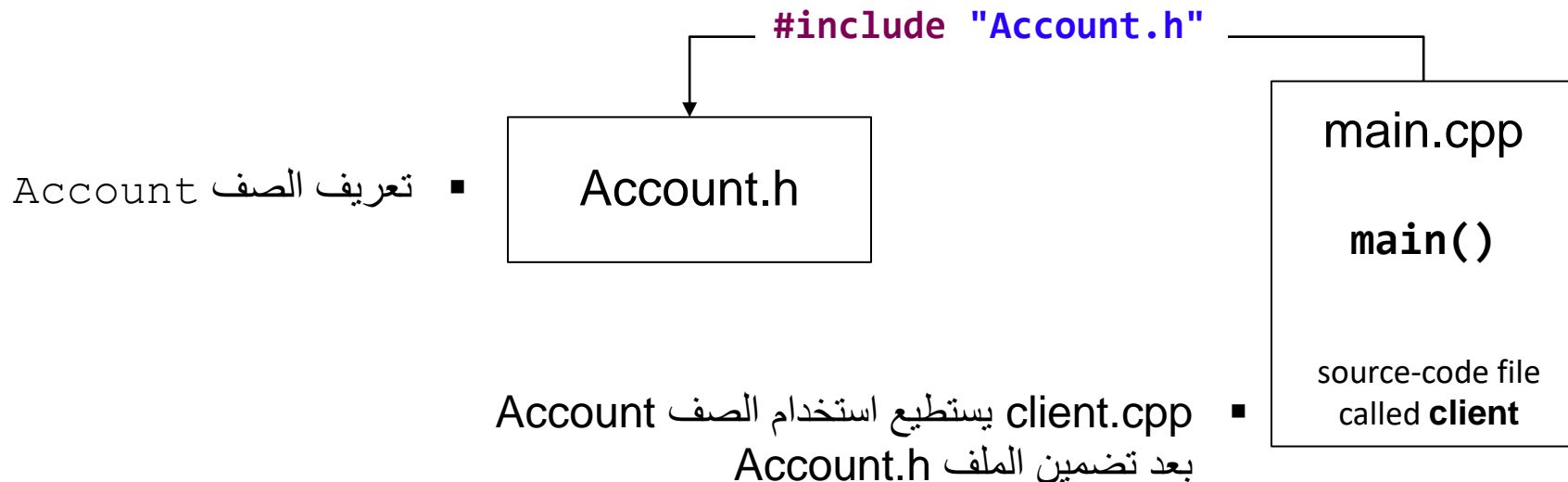
- تعريف الصف في ملف بالإمتداد `.h`. يسمى ذلك الملف بالملف الرأسى (header file)

– `Account.h` يحوي تعريف الصف `Account` فقط

– `main.cpp` يستطيع استخدام الصف `Account` بعد تضمين الملف `Account.h` باستخدام

الموجه `#include`

- يدعم فصل الصف في ملف مبدأ إعادة الإستخدام (reuse)



```
// file's name: Account.h
#include <string>
class Account {
public:
    Account(std::string accountName)
        : name{accountName} { }

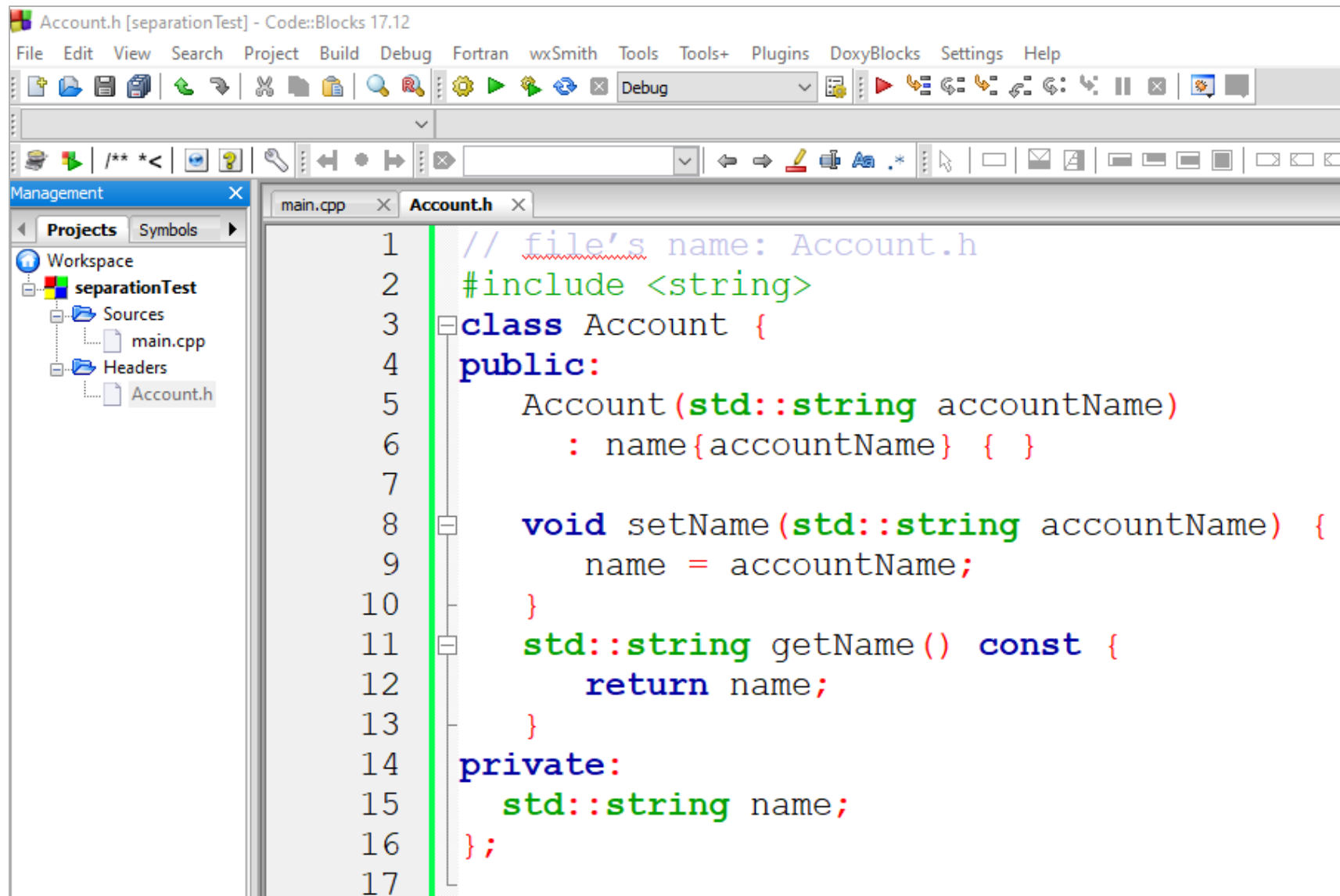
    void setName(std::string accountName)
        name = accountName;
}
std::string getName() const {
    return name;
}
private:
    std::string name;
};
```

```
// file's name: main.cpp
#include <iostream>
#include "Account.h"
using namespace std;

int main() {
    Account account1{"Jane Green"};
    Account account2{"John Blue"};
    cout << "account1 name is: " <<
        account1.getName() << endl;
    cout << "account2 name is: " <<
        account2.getName() << endl;
}
```

Output:

```
account1 name is: Jane Green
account2 name is: John Blue
```

The screenshot shows the Code::Blocks 17.12 IDE with the 'separationTest' project open. The 'Account.h' header file is selected in the 'Headers' folder of the project tree. The code in the editor is as follows:

```
1 // file's name: Account.h
2 #include <string>
3 class Account {
4 public:
5     Account(std::string accountName)
6         : name{accountName} { }
7
8     void setName(std::string accountName) {
9         name = accountName;
10    }
11    std::string getName() const {
12        return name;
13    }
14 private:
15     std::string name;
16 };
17
```

المشيدات الناسخة (copy constructors)

- المشيد الناسخ هو مشيد يقوم بتهيئة غرض باستخدام غرض آخر من نفس الصف

`class_name (const class_name&)`

- يتم استدعاء المشيد الناسخ عند إنشاء (أو خلق) غرض جديد من غرض موجود
- في حال عدم تعريف مشيد ناسخ, يقوم المترجم بإنشاء مشيد ناسخ افتراضي, حيث يقوم ذلك المشيد بنسخ المعطيات الأعضاء بين الأغراض
- يجب تعريف مشيد ناسخ في حال الأغراض التي تحتوي مؤشرات (see slides 79, 80)

المشيدات الناسخة (copy constructors) – An Example

Case #1

يتم استدعاء المشيد الناسخ عند التصريح عن غرض واستخدام معامل الإسناد معاً

```
class Line {
    int _length;
public:
    Line(int length) {
        cout << "Normal constructor" << endl;
        _length = length;
    }
    Line(const Line &obj) {
        cout << "Copy constructor" << endl;
        _length = obj._length;
    }

    void setLength(int length) {
        _length = length;
    }
    int getLength() {
        return _length;
    }
};
```

```
// Main function for the program
int main() {
    Line line1(10);
    Line line2 = line1;
    cout << "Length of line2 : "
         << line2.getLength() << endl;
}
```

Output:

```
Normal constructor
Copy constructor
Length of line2 : 10
```

المشيّدات الناسخة (copy constructors) – An Example

Case #2

يتم استدعاء المشيّد الناسخ عند تمرير
غرض إلى تابع كقيمة

```
class Line {
    int _length;
public:
    Line(int length) {
        cout << "Normal constructor" << endl;
        _length = length;
    }
    Line(const Line &obj) {
        cout << "Copy constructor" << endl;
        _length = obj._length;
    }

    void setLength(int length) {
        _length = length;
    }
    int getLength() {
        return _length;
    }
};
```

Output:

```
Normal constructor
Copy constructor
Length of line : 10
```

```
void display(Line obj);

// Main function for the program
int main() {
    Line line(10);
    display(line);
}

void display(Line obj) {
    cout << "Length of line : "
         << obj.getLength() << endl;
}
```

Copy constructor vs Assignment Operator

- --RECALL-- يتم استدعاء المشيّد الناسخ عند إنشاء (أو خلق) غرض جديد من غرض موجود
- يتم استدعاء معامِل الإسناد عندما يتم إسناد غرض موجود إلى غرض آخر موجود (تمت تهيئته بشكل مسبق)
- إذا لم يتم تعريف معامِل الإسناد ضمن الصف* , يقوم المترجم بإنشاء معامِل إسناد ليقوم بنسخ الغرض بشكل سطحي (shallow copy)

```
...  
int main() {  
    Line line1(10), line2(5);  
    Line line3 = line1; // copy constructor is called  
    line2 = line1; // Assignment Operator is called.  
}  
...
```

*سيتم مناقشة هذا الموضوع لاحقاً

Shallow Copy

Default copy constructor

```
class Line {  
    int _length;  
public:  
    Line(int length) {  
        cout << "Normal constructor" << endl;  
        _length = length;  
    }  
    void setLength(int length) {  
        _length = length;  
    }  
  
    int getLength() {  
        return _length;  
    }  
};
```

يتم استدعاء المشيّد الناسخ الافتراضي
إذا لم يتم تعريف مشيّد ناسخ

```
// Main function for the program  
int main() {  
    Line line1(10);  
    Line line2 = line1;  
    cout << "Length of line2 : "  
        << line2.getLength() << endl;  
}
```

Output:
Normal constructor
Length of line2 : 10

Default copy constructor

- المشيدّ الناسخ الافتراضي لن يعمل بشكل صحيح في حال تضمن الغرض مؤشرات

```
class Line {  
    int* _length;  
public:  
    Line(int length) {  
        cout << "Normal constructor" << endl;  
        _length = new int (length);  
    }  
    void setLength(int length) {  
        *_length = length;  
    }  
    int getLength() {  
        return *_length;  
    }  
};
```

يجب تعريف مشيدّ ناسخ في حال
تضمن الغرض مؤشرات!
(see next slide)

Output:
Normal constructor
Length of line2 : **7**

```
// Main function for the program  
int main() {  
    Line line1(10);  
    Line line2 = line1;  
    line1.setLength(7);  
    cout << "Length of line2 : "  
        << line2.getLength() << endl;  
}
```

- المشيد الناسخ الافتراضي في المثال السابق سيعمل بشكل مشابه للمشيد الناسخ المعروف كمايلي:

```
class Line {  
    int* _length;  
public:  
    Line(int length) {  
        cout << "Normal constructor" << endl;  
        _length = new int (length);  
    }  
    Line(const Line &obj) {  
        cout << "Copy constructor" << endl;  
        _length = obj._length;  
    }  
  
    void setLength(int length) {  
        *_length = length;  
    }  
    int getLength() {  
        return *_length;  
    }  
};
```

```
// Main function for the program  
int main() {  
    Line line1(10);  
    Line line2 = line1;  
    line1.setLength(7);  
    cout << "Length of line2 : "  
        << line2.getLength() << endl;  
}
```

Output:

Normal constructor
Copy constructor
Length of line2 : 7

ما هو الحل؟

```
_length = new int(*obj._length);
```

Then, the Output:

Normal constructor
Copy constructor
Length of line2 : 10

class_name (const class_name&)

- لماذا يجب تمرير الوسيط إلى المشيّد الناسخ باستخدام التمرير بالمرجع؟

– باعتبار مايلي:

- يتم استدعاء المشيّد الناسخ عندما يتم تمرير غرض كقيمة إلى تابع

- المشيّد الناسخ هو تابع

❖ لذلك, إذا تم تمرير قيمة ضمن المشيّد الناسخ, يتم استدعاء المشيّد الناسخ ويتم الدخول في سلسلة

لانهائية من الاستدعاءات

```
...  
Line(const Line obj) {  
    cout << "Copy constructor" << endl;  
    _length = obj._length;  
}  
...
```

**error: invalid constructor; you probably meant
'Line (const Line&)'**

`class_name (const class_name&)`

- لماذا يجب أن يكون الوسيط (الغرض) المُرر إلى المَشِيّد الناسخ ثابت؟

1. حتى لا يتم تعديل الغرض بشكل غير مقصود ضمن المَشِيّد الناسخ

2. يوجد سبب آخر عند إعادة غرض بقيمة ومن ثم استدعاء المَشِيّد الناسخ. (see next slide)

```

class Line {
    int _length;
public:
    Line(int length) {
        cout << "Normal constructor" << endl;
        _length = length;
    }
    Line(Line &obj) {
        cout << "Copy constructor" << endl;
        _length = obj._length;
    }
    int getLength() {
        return _length;
    }
};

Line f(){
    Line line1(10);
    return line1;
}

int main() {
    Line line2 = f();
    cout << "Length of line2 : "
        << line2.getLength() << endl;
}

```

عند إعادة غرض بقيمة من التابع f, يقوم المترجم بإنشاء غرض بشكل مؤقت والذي بدوره يُنسخ إلى الغرض line2 باستخدام المشيّد الناسخ.

حذف const في المشيّد الناسخ, سيعطي خطأ ترجمة, لأن الأغراض التي يتم إنشاؤها بشكل مؤقت لا يمكن ربطها بمراجع غير ثابتة.

error: cannot bind non-const lvalue reference of type 'Line&' to an rvalue of type 'Line'

```


class Line {
    int _length;
public:
    Line(int length) {
        cout << "Normal constructor" << endl;
        _length = length;
    }
    Line(const Line &obj) {
        cout << "Copy constructor" << endl;
        _length = obj._length;
    }
    int getLength() {
        return _length;
    }
};

Line f(){
    Line line1(10);
    return line1;
}

int main() {
    Line line2 = f();
    cout << "Length of line2 : "
        << line2.getLength() << endl;
}

```

Output:
Normal constructor
Length of line2 : 10



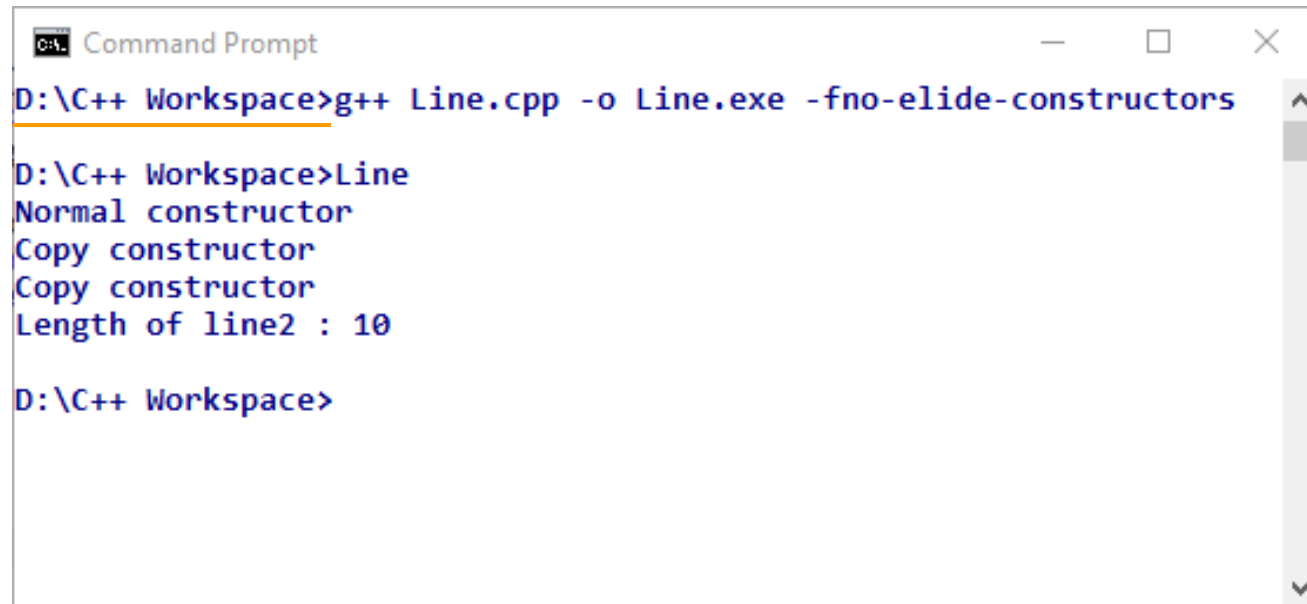
ما هو سبب عدم استدعاء المشيّد الناسخ؟
السبب هو أن معظم المترجمات تقوم بعملية
تحسين لتجنب العمليات الغير ضرورية
لنسخ الأغراض.

خيار الترجمة `-fno-elide-constructors`

-- للإطلاع! --

- من أجل تجاهل التحسين السابق الذي يقوم به المترجم والذي يتم من خلاله تجنب العمليات الغير ضرورية لنسخ الأغراض, يمكن إضافة الخيار `-fno-elide-constructors`
- ترجمة وربط برنامج C++ السابق, بفرض اسم الملف المصدر هو `Line.cpp`

```
g++ Line.cpp -o Line.exe -fno-elide-constructors
```



```
Command Prompt
D:\C++ Workspace>g++ Line.cpp -o Line.exe -fno-elide-constructors
D:\C++ Workspace>Line
Normal constructor
Copy constructor
Copy constructor
Length of line2 : 10
D:\C++ Workspace>
```

Using GNU C++ compiler

- يمكن للمشيّد الناسخ أن يكون خاص ضمن الصف (محدد الوصول private)
- في هذه الحالة تصبح الأغراض من ذلك الصف غير قابلة للنسخ (non-copyable)
- عندما يحتوي الصف مؤشرات أو متحولات محجوزة ديناميكياً،
 - يمكن إما كتابة مشيّد ناسخ يراعي المؤشرات (نسخ عميق deep copy) (انظر المثال: slide 80), أو
 - تعريف المشيّد الناسخ ليكون خاص، وبالتالي يقوم المترجم بتوليد خطأ ترجمة عند أي محاولة لنسخ الغرض.

قواعد مجالات الرؤية Scope Rules

- مجال الرؤية لمتحول ما هو جزء البرنامج حيث يمكن استخدام ذلك المتحول. يوجد خمس حالات:
 - مجال رؤية على مستوى الكتلة (Block Scope)
 - مجال رؤية على مستوى التابع (Function Scope)
 - مجال رؤية عام (Global Scope)
 - مجال رؤية على مستوى نموذج التابع (Function-Prototype Scope)
 - مجال رؤية على مستوى الصف (Class Scope)

! فترة التخزين (storage duration) ليس لها تأثير على مجال رؤية المتحولات

```
class MyClass {
    int x = 1;
public:
    void displayX() {
        int x = 2;
        // using scope resolution operator ::
        cout << "x= " << MyClass::x << endl;
    }
};

int x = 5; // global x
int main() {
    int x = 7; // local x
    cout << x << endl;

    // using scope resolution operator ::
    cout << ::x << endl;

    MyClass obj;
    obj.displayX();

    for (;;) {
        static int i = 0;
        cout << i++ << "\n";
        if (i > 5)
            goto exit;
    }
    // function scope
    exit: cout << "exit" << endl;
}
```

7

5

x= 1

0

1

2

3

4

5

exit

تمرين #2

```
void func1() {
    int x = 25;
    cout << "X= " << x++ << endl;
    cout << "X= " << x << endl;
}

void func2() {
    static int x = 50;
    cout << "X= " << ++x << endl;
    cout << "X= " << x << endl;
}

int x = 1; // global variable
```

```
int main() {
    cout << "X= " << x << endl; ..... X=1
    int x = 5; // local variable to main
    cout << "X= " << x << endl; ..... X=5
    {
        int x = 7;
        cout << "X= " << x << endl; ..... X=7
    }
    cout << "X= " << x << endl; ..... X=5
    func1(); ..... X=25
    func2(); ..... X=51 X=26
    func1(); ..... X=25 X=51
    func2(); ..... X=52 X=26
    cout << "X= " << x << endl; ..... X=5 X=52
}
```

```
int main() {  
  
    int arr[] = { 3, 1, 5, 9, 10 };  
  
    int *aPtr = arr;  
    aPtr[3] = 20;  
    aPtr+=2;  
    cout << "#1: " << * (--aPtr) << endl;  
    cout << "#2: " << *(arr + 2) << endl;  
    cout << "#3: " << aPtr[1] << endl;  
  
    int a[3][3] = { };  
}
```

1. أوجد خرج البرنامج

2. مستخدماً المؤشرات،
اكتب التعليمات اللازمة
لطباعة عنوان وقيمة
العنصر a[2][2]

3. عرف مصفوفة ثنائية b مؤلفة من ثلاث أسطر وأربع أعمدة ليتم حجز مواقع عناصرها في الذاكرة بشكل ديناميكي وتهيئة عناصرها بقيم صفرية. ثم اطبع قيم عناصرها مستخدماً المؤشرات.

```
int main() {
```

```
...
```

```
int a[3][3] = { };
```

```
cout << &a[2][2] << endl;
cout << *a + 6 + 2 << endl;
cout << *(a + 2) + 2 << endl;
```

```
cout << a[2][2] << endl;
cout << *(*a + 6 + 2) << endl;
cout << *(* (a + 2) + 2) << endl;
```

```
int **b = new int*[3];
for (int i = 0; i < 3; i++)
    *(b + i) = new int[4] { };
```

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; j++)
        cout << *(* (b + i) + j) << endl;
```

```
}
```

.1

output#1: 1

output#2: 5

output#3: 5

.2

.3

- توابع inline هامة بشكل عام عندما يكون زمن استدعاء تابع أكبر من زمن تنفيذه:
 $\text{function calling time} > \text{function execution time}$
- الكلمة المفتاحية inline **تطلب** من المترجم توليد نسخة من جسم التابع في كل مكان يستدعى فيه التابع لتجنب استدعاء التابع

```
inline double getCubeVolume(double side) {  
    return side * side * side; // calculate cube's volume  
}  
  
int main() {  
    double sideValue;  
    cout << "Enter the side length of your cube: ";  
    cin >> sideValue;  
  
    cout << "Volume of cube with side" << sideValue << " is "  
        << getCubeVolume(sideValue) << endl;  
}
```

- جميع التوابع الأعضاء في صف هي inline بشكل افتراضي

الوسطاء الافتراضية (default arguments)

- يمكن تعريف وسطاء افتراضية عند التصريح عن تابع
- عند استدعاء التابع بوسطاء افتراضية, يقوم المترجم بإعادة كتابة استدعاء التابع وإدخال القيم الافتراضية لهذه الوسطاء

```
int sum(int x, int y, int z=0, int w=0) {  
    return (x + y + z + w);  
}  
  
int main() {  
    cout << sum(10, 15) << endl;  
    cout << sum(10, 15, 25) << endl;  
    cout << sum(10, 15, 25, 30) << endl;  
    return 0;  
}
```

Output:

25
50
80

! الإنتباه عند استخدام التحميل الزائد للتوابع مع استخدام الوسطاء الافتراضية

```
int sum(int x, int y, int z=0, int w=0) {  
    return (x + y + z + w);  
}  
  
int sum(int x, int y, float z=0, float w=0) {  
    return (x + y + z + w);  
}  
  
int main() {  
    cout << sum(10, 15) << endl;  
    cout << sum(10, 15, 25) << endl;  
    cout << sum(10, 15, 25, 30) << endl;  
    return 0;  
}
```

error: call of overloaded 'sum(int, int)' is ambiguous|

التوابع الصديقة (friend functions)

- التوابع الصديقة ليست توابع أعضاء! يُصرح عنها باستخدام الكلمة المفتاحية friend ويمكنها الوصول إلى أعضاء (التوابع والمعطيات) الصف العامة والخاصة، ولكن ليس بشكل مباشر!
- ليس لها علاقة بمحددات الوصول، لذلك يمكن تعريفها في أي مكان ضمن الصف!

```
class Counter {  
    friend void setCounter( Counter &, int ); // friend declaration  
public:  
    Counter() : count( 0 ) {}
```

```
    void print() const {  
        cout << count << endl;  
    }  
private:  
    int count; // data member  
};
```

```
int main() {  
    Counter counter;  
    // set count using a friend function  
    setCounter(counter, 8);  
    counter.print();  
}
```

```
// function setCounter can modify private data of Count because  
// setCounter is declared as a friend of Count  
void setCounter( Counter &c, int val ) {  
    c.count = val; // allowed because setCounter is a friend of Counter  
}
```

- لدى كل غرض إمكانية الوصول إلى عنوانه بواسطة مؤشر يسمى this
- المؤشر this ليس جزء من الغرض نفسه, ولكن يقوم المترجم بتمريره كوسيط ضمني لكل من التوابع الأعضاء غير الساكنة
- الاستخدام الشائع للمؤشر this هو لتجنب تعارض الأسماء (naming conflicts) بين المعطيات الأعضاء لصف و وسطاء التوابع الأعضاء (أو المتحولات المحلية الأخرى). مثال؟
- يمكن استخدام this لتمكين الاستدعاء المتتالي للتوابع الأعضاء

استخدام this لتمكين الاستدعاء المتتالي للتوابع الأعضاء 2 / 1

```
class Point {
public:
    Point& setX(int x) {
        this->x = x;
        return *this;
    }
    Point& setY(int y) {
        this->y = y;
        return *this;
    }
    void print() {
        cout << "( " << x << " , " << y << " )";
    }
private:
    int x, y;
};
```

Case #1

إعادة الغرض (كمرجع) الذي يشير
له المؤشر this

```
int main() {
    Point P1;
    P1.setX(3).setY(5).print();
}
```

Output:
(3 , 5)

استخدام this لتمكين الاستدعاء المتتالي للتوابع الأعضاء 2 / 2

Case #2

إعادة مؤشر إلى الغرض

```
class Point {  
public:  
    Point* setX(int x) {  
        this->x = x;  
        return this;  
    }  
    Point* setY(int y) {  
        this->y = y;  
        return this;  
    }  
    void print() {  
        cout << "( " << x << " , " << y << " )";  
    }  
private:  
    int x, y;  
};
```

```
int main() {  
    Point P1;  
    P1.setX(3)->setY(5)->print();  
}
```

Output:
(3 , 5)

استدعاء التوابع الأعضاء من مؤشر إلى غرض

```
int main() {  
  
    Point point; // an point object  
  
    Point& pointRef = point; // pointRef refers to an Point object  
  
    Point* pointPtr = &point; // pointRef points to an Point object  
  
    point.setX( 5 ); // call setX via the Point object  
  
    pointRef.setX( 5 ); // call setX via a reference to the Point object  
  
    pointPtr-> setX( 5 ); // call setX via a pointer to the Account object  
  
    (*pointPtr).setX( 5 ); // call setX via a pointer to the Account object  
  
}
```

- المتحولات الساكنة (static variables) -- RECALL –
- يمكن أيضاً تعريف المعطيات والتوابع الأعضاء في صف لتكون ساكنة (static members of Class)
 - Static data members
 - Static member functions
- تتم مشاركة المعطيات والتوابع الأعضاء الساكنة لجميع الأغراض التي يتم إنشاؤها من صف

2 / 1 Static data members

```
class Account {
    string name;
    int id;
    static int count;
public:
    Account(string accountName) : name { accountName } {
        id= ++count;
        cout << "Account #" << id << " is created" << endl;
    }
    int getID() {
        return id;
    }
    string getName() {
        return name;
    }
};
```

```
int Account::count = 0;
int main() {
    Account accounts[2] =
        {Account("Jane Green"), Account("John Blue")};
    for (int i = 0; i < 2; i++)
        cout << "ID: " << accounts[i].getID()
            << "   Name: " << accounts[i].getName()
            << endl;
}
```

Output:

```
Account #1 is created
Account #2 is created
ID: 1   Name: Jane Green
ID: 2   Name: John Blue
```

2 / 2 Static data members

- المعطيات الأعضاء الساكنة يجب أن يُعاد التصريح عنها وثم تهيئها خارج تعريف الصف بدون استخدام الكلمة المفتاحية `static` (القيمة الافتراضية بدون تهيئة 0)

- يتم إعادة التصريح عنها باستخدام المعامل `::` (scope resolution operator) لتحديد الصف التي تنتمي إليه

```
int Account::count = 0;
```

- حالة خاصة: يجب تهيئة المعطيات الساكنة الثابتة عند التصريح عنها ضمن الصف

```
static const int x = 0;
```

- يوجد نسخة واحدة من المعطيات الأعضاء الساكنة لكل صف (أي نسخة مشتركة لجميع أغراض الصف)

Static member functions

- التوابع الأعضاء الساكنة تستطيع الوصول إلى المعطيات أو التوابع الأعضاء الساكنة فقط
- يتم استدعاء التوابع الأعضاء باستخدام غرض من الصف أو باستخدام الصف مباشرة

className::memberFunctionName (...);

```
class Account {  
    string name;  
    int id;  
    static int count;  
public:  
    Account(string accountName):  
        name { accountName} {  
        id = ++count;  
        cout << "Account #" << id  
        << " is created" << endl;  
    }  
    static int getCount() {  
        return count;  
    }  
};
```

Output:

```
Account #1 is created  
Account #2 is created  
The number of created objects is 2
```


```
int Account::count = 0;  
int main() {  
    Account accounts[2] =  
        {Account("Jane Green"),  
         Account("John Blue") };  
    cout << "The number of created Accounts is "  
        << Account::getCount() << endl;  
}
```

الأغراض الساكنة static objects

```
class A {
    int x;
public:
    A(){cout << "A's constructor" << endl;}
};

class B {
    static A a; // declaration of a
public:
    B(){cout << "B's constructor" << endl;}
    static A getA() {
        return a;
    }
};
```


- لا يتم استدعاء المَشِيد عند التصريح عن غرض ضمن المعطيات الأعضاء لصف باستخدام الكلمة المفتاحية static. يتم استدعاء المَشِيد في حال تعريف ذلك الغرض خارج الصف



```
int main() {
    B b1, b2, b3;
}
```


Output:

B's constructor
B's constructor
B's constructor



```
int main() {
    B b1, b2, b3;
    A a1 = b1.getA();
}
```

undefined reference to `B::a'



```
// definition of a
A B::a;

int main() {
    B b1, b2, b3;
    A a1 = b1.getA();
}
```

Output:

A's constructor
B's constructor
B's constructor
B's constructor

التحميل الزائد للمعاملات (operator overloading)

- C++ لا تسمح بإنشاء معاملات (operators) جديدة! لكنها تسمح بتحميل المعاملات الموجودة بشكل زائد
- تطبيق المعاملات في C++ على أغراض صف, يسمى التحميل الزائد للمعاملات بحيث يصبح لهذه المعاملات معنى مناسب لهذه الأغراض
- عملية التحميل الزائد ليست أوتوماتيكية! يجب كتابة تابع لهذه العملية
- مثلاً لتحميل معامل الجمع, يجب كتابة تابع اسمه `operator+`
- عندما تحمل المعاملات كتوابع أعضاء لصف, يجب أن لا تكون `static`! لأنه يجب استدعاؤها من غرض لصف لتنفيذ على ذلك الغرض

- أسبقية المعاملات لا يمكن تغييرها
 - لا يمكن تغيير معامل أحادي إلى معامل ثنائي, أو معامل ثنائي إلى أحادي
 - معنى المعامل يجب أن لا يتغير عند تحميله بشكل زائد
 - معاملات لا يمكن تحميلها بشكل زائد:
- . .*(pointer to member) :: ? :

التحميل الزائد لمعاملات على الصف string

- مثال الصف string – لاستخدام الصف string `#include <string>`

```
int main() {
    string s1; // uses default string constructor
               // to create an empty string
    string s2 = "Testing Operator Overloading";
    s1=s2; // overloading = operator
    s1+= " with class string!"; // overloading += operator
    cout << s1 << endl;
    // string class has member functions like empty and substr
    if (s1.empty())
        cout << "s1 string is empty";
    else cout << s1.substr(29) <<endl;

    cout << s1.substr(0,7)<<endl;
    cout << s1[0] <<endl; // no bounds checking
    cout << s1.at(0) <<endl; // exception can be handled!
    cout << s1.length();
}
```

Output:

```
Testing Operator Overloading with class string!
with class string!
Testing
T
T
47
```

- يمكن تحميل المعاملات الثنائية بحالتين:

– كتابع عضو غير ساكن non-static member function لديه وسيط واحد

```
class Counter {  
public:  
    Counter(int);  
    bool operator<(Counter &);  
private:  
    int count;  
};
```

– كتابع غير عضو non-member function لديه وسيطين, بحيث أحد هذين الوسيطين يجب أن يكون غرض الصف أو مرجع إلى غرض الصف

```
friend bool operator<(Counter &a, Counter &b);
```

As non-static member function

```
class Counter{
private:
    int count;
public:
    Counter(int x) {
        count = x;
    }
    bool operator<(Counter &c) {
        if (this->count < c.count)
            return true;
        else
            return false;
    }
};
```

```
int main() {
    Counter counter1(3);
    Counter counter2(5);
    // if (counter1.operator<(counter2)) {
    if (counter1<counter2) {
        cout<< "counter1<counter2";
    }
    else
        cout << "counter1>=counter2";
}
```

As non-member function

```
class Counter {  
private:  
    int count;  
public:  
    Counter(int x) {  
        count = x;  
    }  
    friend bool operator<(Counter&, Counter&);  
};
```

```
bool operator<(Counter &a, Counter &b) {  
    if (a.count < b.count)  
        return true;  
    else  
        return false;  
}
```

```
int main() {  
    Counter counter1(3);  
    Counter counter2(5);  
    //or if (operator<(counter1, counter2)) {  
    if (counter1 < counter2) {  
        cout << "counter1 < counter2";  
    }  
    else  
        cout << "counter1 >= counter2";  
}
```

مثال على التحميل الزائد للمعامل << والمعامل >>

```
class Point {
    friend ostream& operator<<(ostream&, Point&);
    friend istream& operator>>(istream&, Point&);
private:
    int x_, y_;
};

ostream& operator<<(ostream &output, Point &p) {
    output << "P(" << p.x_ << "," << p.y_ << ")";
    return output;
}

istream& operator>>(istream &input, Point &p) {
    cout << "Please enter x: ";
    input >> p.x_;
    cout << "Please enter y: ";
    input >> p.y_;

    return input;
}
```

```
main()
{
    Point p1;
    cin>> p1;
    cout<< "The Point entered is: "<< p1;
}
```

- يمكن تحميل المعاملات الأحادية بحالتين:

– كتابع عضو غير ساكن non-static member function بدون وسطاء

```
class Counter
{
public:
    Counter& operator++();
private:
    int count;
};
```

– كتابع غير عضو non-member function بوسيط واحد (غرض الصف أو مرجع إلى غرض الصف)

```
friend Counter& operator++(Counter &);
```


مثال على التحميل الزائد للمعامل ++ (سابق)

```
class Counter {  
  
public:  
    Counter& operator++() {  
        ++count;  
        return *this;  
    }  
    void display() {  
        cout << "count = "  
            << count << endl;  
    }  
private:  
    int count = 3;  
};
```

```
int main() {  
    Counter counter1;  
  
    cout << "Before increment: ";  
    counter1.display();  
  
    Counter counter2 = ++counter1;  
    cout << "counter2 after pre increment: ";  
    counter2.display();  
    cout << "counter1 after pre increment: ";  
    counter1.display();  
}
```

Output:

```
Before increment: count = 3  
counter2 after pre increment: count = 4  
counter1 after pre increment: count = 4
```

مثال على التحميل الزائد للمعامل ++ (لاحق)

```
class Counter {  
  
public:  
    Counter operator++(int) {  
        Counter temp;  
        temp.count = count++;  
        return temp;  
    }  
    void display() {  
        cout << "count = "  
            << count << endl;  
    }  
private:  
    int count = 3;  
};
```

```
int main() {  
    Counter counter1;  
  
    cout << "Before increment: ";  
    counter1.display();  
  
    Counter counter2 = counter1++;  
    cout << "counter2 after post increment: ";  
    counter2.display();  
    cout << "counter1 after post increment: ";  
    counter1.display();  
}
```

Output:

```
Before increment: count = 3  
counter2 after post increment: count = 3  
counter1 after post increment: count = 4
```

- الوراثة هي شكل من أشكال إعادة الإستخدام (reuse), حيث يمكن إنشاء صف جديد يمكنه أخذ خصائص صف موجود (existing class), ومن ثم تخصيصها (customize)
- يسمى ذلك الصف الموجود بـ base class, ويسمى الصف الجديد بـ derived class
- أمثلة:

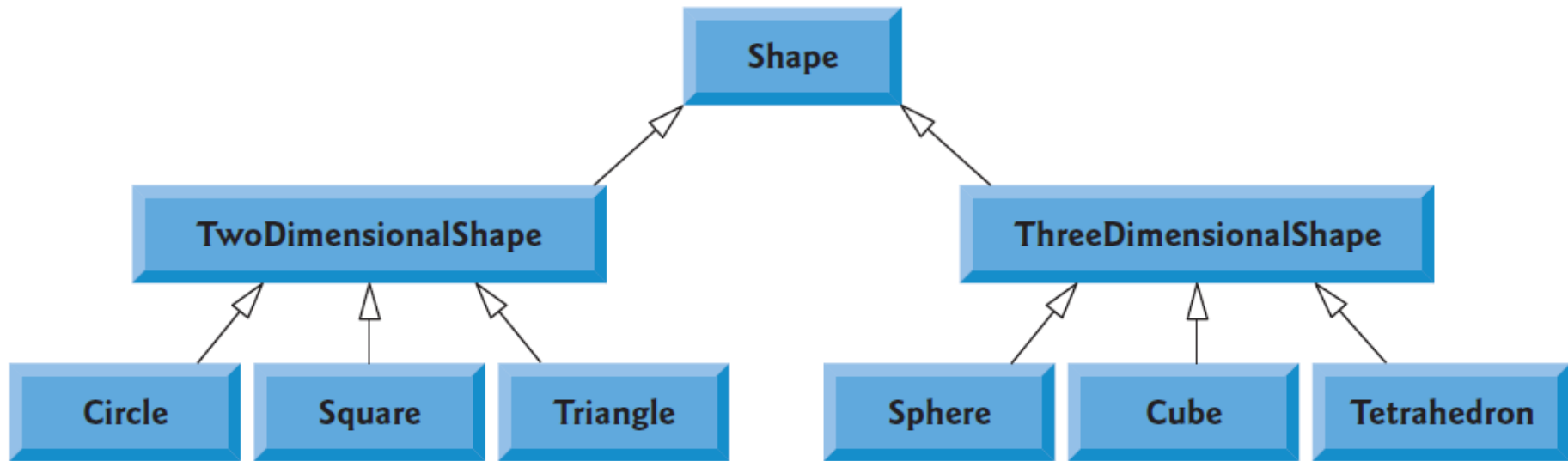
Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

The name of new Class

The name of Existing Class

```
class derived-class: access-specifier base-class {  
    public, protected, or private  
};
```

```
class base-class  
{  
    //Body of parent class  
};  
  
class derived-class : access_modifier base-class  
{  
    //Body of child class  
};
```



```
class TwoDimensionalShape : public Shape  
class Circle : public TwoDimensionalShape  
...
```

- يمكن توريث صف (base class) لصف آخر (derived class) من خلال ثلاث طرق حسب محددات الوصول:

محددات الوصول في الـ base class	أنواع الوراثة		
	Public Inheritance	Protected Inheritance	Private Inheritance
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Base-class member accessibility in a derived class

```
class A {  
public:  
    int x;  
protected:  
    int y;  
private:  
    int z;  
};
```

```
class B : public A {  
    // x is public  
    // y is protected  
    // z is not accessible from B  
};
```

- الصف B يرث الصف باستخدام محدد الوصول **public**
 - الأعضاء العامة في الصف A تصبح عامة في الصف B
 - الأعضاء المحمية في الصف A تصبح محمية في الصف B
 - الصف B لا يمكنه الوصول إلى الأعضاء الخاصة في الصف A

Protected Inheritance الوراثة المحمية

```
class A {  
public:  
    int x;  
protected:  
    int y;  
private:  
    int z;  
};
```

```
class B : protected A {  
    // x is protected  
    // y is protected  
    // z is not accessible from B  
};
```

• الصف B يرث الصف باستخدام محدد الوصول **protected**

- الأعضاء العامة في الصف A تصبح محمية في الصف B
- الأعضاء المحمية في الصف A تصبح محمية في الصف B
- الصف B لا يمكنه الوصول إلى الأعضاء الخاصة في الصف A

الوراثة الخاصة Private Inheritance

```
class A {  
public:  
    int x;  
protected:  
    int y;  
private:  
    int z;  
};
```

- الصف B يرث الصف باستخدام محدد الوصول **private**
 - الأعضاء العامة في الصف A تصبح خاصة في الصف B
 - الأعضاء المحمية في الصف A تصبح خاصة في الصف B
 - الصف B لا يمكنه الوصول إلى الأعضاء الخاصة في الصف A

```
class B : private A { // 'private' is default for classes  
    // x is private  
    // y is private  
    // z is not accessible from B  
}
```

```
class Person {  
public:  
    Person(string name = "", int age = 0);  
    void setName(string);  
    string getName() const;  
    void setAge(int);  
    int getAge() const;  
  
private:  
    string name_;  
    int age_;  
  
};
```

Person
- name_: String - age_: int
+ Person(name:String, age:int) + setName(name: String): void + getName(): String + setAge(age:int) : void + getAge(): int

```
class Employee {
public:
    Employee(string name = "", int age = 0, double hourlySalary = 0.0,
              long employeeID = 0);
    void setName(string);
    string getName() const;
    void setAge(int);
    int getAge() const;
    void setHourlySalary(double);
    void displayEmployeeInfo() const;

Private:
    string name_;
    int age_;
    double hourlySalary_;
    long employeeID_;
};
```

Employee
<ul style="list-style-type: none">- name_: String- age_: int- hourlySalary_:double- employeeID_:long
<ul style="list-style-type: none">+ Employee(name:String, age:int, hourlySalary:double, employeeID:long)+ setName(name: String): void+ getName(): String+ setAge(age:int) : void+ getAge(): int+ setHourlySalary(hourlySalary: double): void+ displayEmployeeInfo(): void

Person

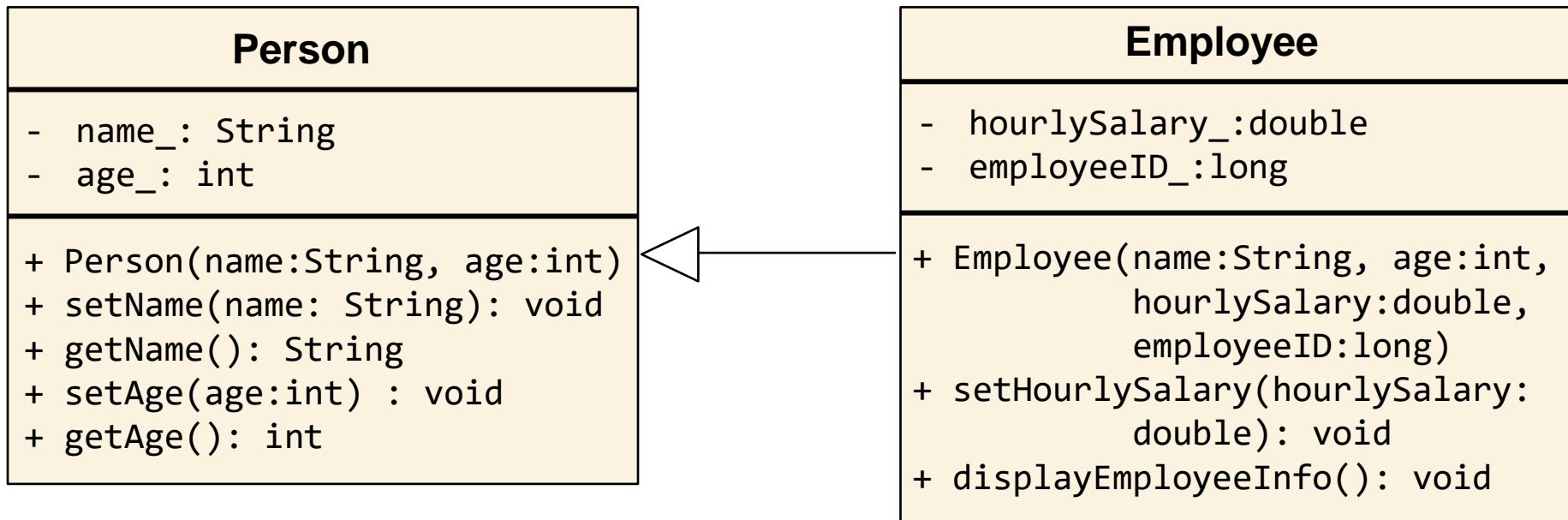
```
- name_: String
- age_: int

+ Person(name:String, age:int)
+ setName(name: String): void
+ getName(): String
+ setAge(age:int) : void
+ getAge(): int
```

Employee

```
- name_: String
- age_: int
- hourlySalary_:double
- employeeID_:long

+ Employee(name:String, age:int,
            hourlySalary:double,
            employeeID:long)
+ setName(name: String): void
+ getName(): String
+ setAge(age:int) : void
+ getAge(): int
+ setHourlySalary(hourlySalary:
                  double): void
+ displayEmployeeInfo(): void
```



```

class Employee: public Person {
public:
    Employee(string name, int age, double hourlySalary = 0.0,
               long employeeID = 0);
    void setHourlySalary(double);
    void displayEmployeeInfo() const;
Private:
    double hourlySalary_;
    long employeeID_;
};
    
```

مثال على الوراثة 5 / 5

```
class Person{
public:
    Person(string name = "TestName", int age = 0)
        : name_(name), age_(age){}
    void setName(string name){
        name_ = name;
    }
    string getName() const{
        return name_;
    }
    void setAge(int age){
        age_ = age;
    }
    int getAge() const{
        return age_;
    }
private:
    string name_;
    int age_;
};
```

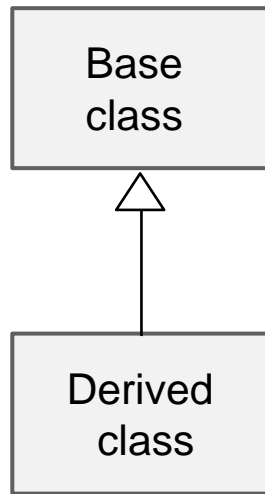
Output:

```
ID: 1
Name: TestName
Age: 0
Salary per hour: 20.25
```

```
class Employee: protected Person{
    double hourlySalary_;
    long employeeID_;
public:
    Employee(double hourlySalary = 0.0,
              long employeeID = 0) :
        hourlySalary_(hourlySalary),
        employeeID_(employeeID){}

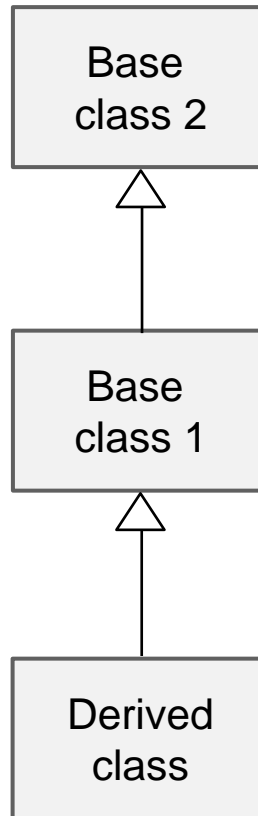
    void getEmployeeInfo() const{
        cout << "ID: " << employeeID_ << endl
              << "Name: " << getName() << endl
              << "Age: " << getAge() << endl
              << "Salary per hour: " << hourlySalary_;
    }
};

int main(){
    Employee emp1(20.25, 1);
    emp1.getEmployeeInfo();
}
```



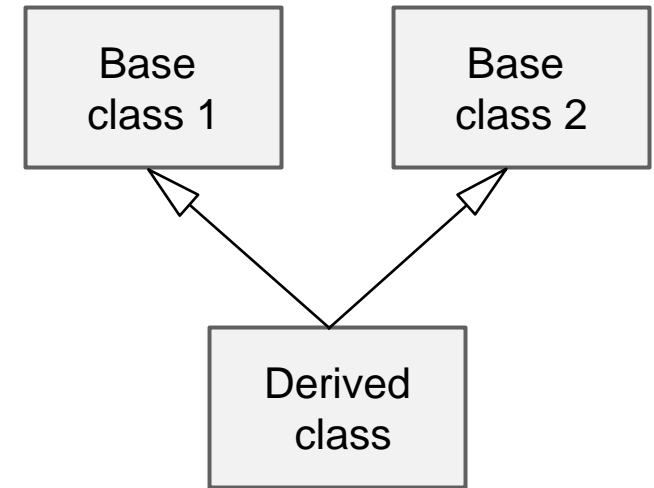
Single Inheritance

يمكن لصف واحد فقط أن يرث
من صف واحد فقط



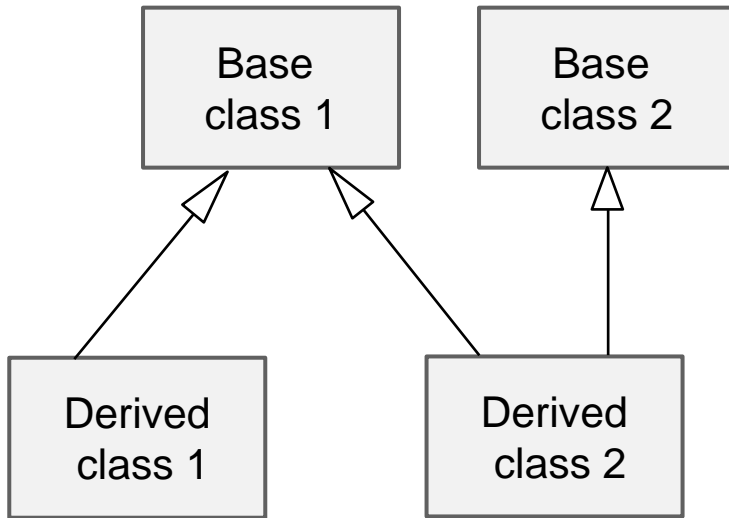
Multilevel Inheritance

يمكن لصف أن يرث من صف,
الذي بدوره يرث من صف آخر



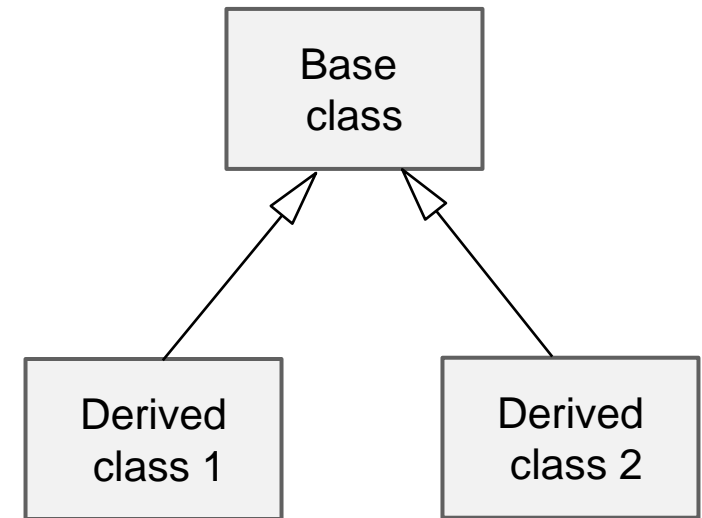
Multiple Inheritance

يمكن لصف أن يرث من أكثر من صف



Hybrid Inheritance

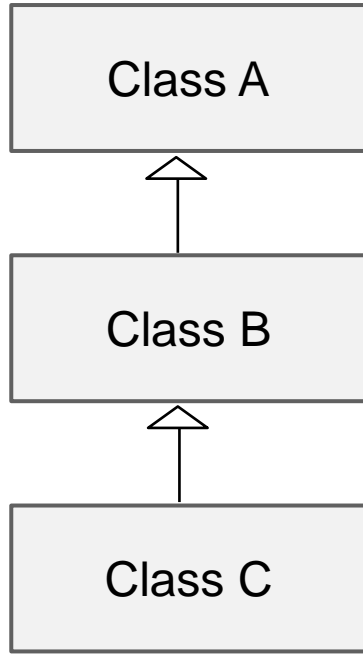
دمج نوعين أو أكثر من أنواع الوراثة
Hierarchical and multiple inheritance



Hierarchical Inheritance

يمكن لأكثر من صف أن يرث من صف واحد

المشيدات في الصفوف المشتقة Derived Classes



- يتم استدعاء مشيد الصف المشتق أولاً, ثم مشيد الصف الأساسي, وهكذا في حال الصف الأساسي هو بدوره مُشتق من صف آخر...

- آخر مشيد يُستدعى هو مشيد الصف الأساسي, ولكن يتم تنفيذه أولاً

- آخر مشيد يتم تنفيذه هو مشيد الصف المشتق

1. Class C constructor is called
2. Class B constructor is called
3. Class A constructor is called
4. Class A constructor' body finishes executing
5. Class B constructor' body finishes executing
6. Class C constructor' body finishes executing

استدعاء مشيد الصف الأساسي من الصف المشتق

```
class A {
    int x_;
public:
    A(int x) {
        x_ = x;
        cout << "A's Constructor executed: Value of x: " << x_ << endl;
    }
};

// Class B is derived from A
class B: A {
public:
    B(int x) : A(x) { //Initializer list must be used
        cout << "B's Constructor executed";
    }
};

int main() {
    B obj(10);
    return 0;
}
```

Output:

A's Constructor executed: Value of x: 10
B's Constructor executed

يمكن استخدام قائمة التهيئة (initializer list) مع
المشيد لتهيئة المعطيات الأعضاء لصف, يوجد حالات
يجب معها استخدام قائمة التهيئة. أمثلة؟

حالات أخرى يجب معها استخدام قائمة التهيئة

تهيئة المعطيات الثابتة غير الساكنة

```
class Test {  
    const int x;  
public:  
    Test(int x) : x(x) {}  
    int getX() {return x;}  
};  
int main() {  
    Test t(3);  
    cout<<t.getX()<<endl;  
}
```

Output:
3

تهيئة المعطيات المصرح عنها كمرجع

```
class Test {  
    int &x;  
public:  
    Test(int &a) : x(a) {}  
    int getX() {return x;}  
};  
int main() {  
    int i = 20;  
    Test t1(i);  
    cout <<t1.getX()<< endl;  
    i = 30;  
    cout <<t1.getX()<< endl;  
}
```

Output:
20
30



استدعاء تابع عضو في الصف الأساسي من تابع عضو في الصف المشتق

```
class A {
public:
    void f() {
        cout << "f() -- A's class " << endl;
    }
};

class B: A {
public:
    // function overriding!
    void f() {
        A::f(); // calling a function from base class
        cout << "f() -- B's class" << endl;
    }
};

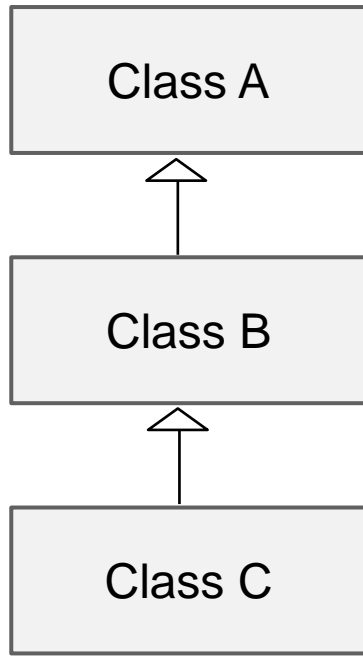
int main() {
    B b;
    b.f();
}
```

عندما يكون التابعين بنفس
الاسم, يُستخدم المعامل ::

Output:

f() -- A's class
f() -- B's class

الهامد في الصفوف المشتقة Derived Classes



- عند تحرير الذاكرة المحجوزة لغرض من صف مشتق, يتم استدعاء هادم الغرض
- عندما يُستدعى هادم غرض من صف مشتق, يتم تنفيذه أولاً ومن ثم استدعاء هادم الغرض في الصف الأساسي. وهكذا في حال الصف الأساسي هو بدوره مُشتق من صف آخر...
- بعد تنفيذ آخر هادم (للصف الأساسي في أعلى الهيكلية), يتم تحرير الذاكرة المحجوزة للغرض

To destroy an object from C:

1. Class C destructor is called and executed
2. Class B destructor is called and executed
3. Class A destructor is called and executed
4. Memory is released

```
class A {
public:
    A() {
        cout << "A's constructor is executed"
              << endl;
    }
    ~A() {
        cout << "A's destructor is executed"
              << endl;
    }
};

class B : public A {
public:
    B() {
        cout << " B's constructor is executed"
              << endl;
    }
    ~B() {
        cout << " B's destructor is executed"
              << endl;
    }
};
```

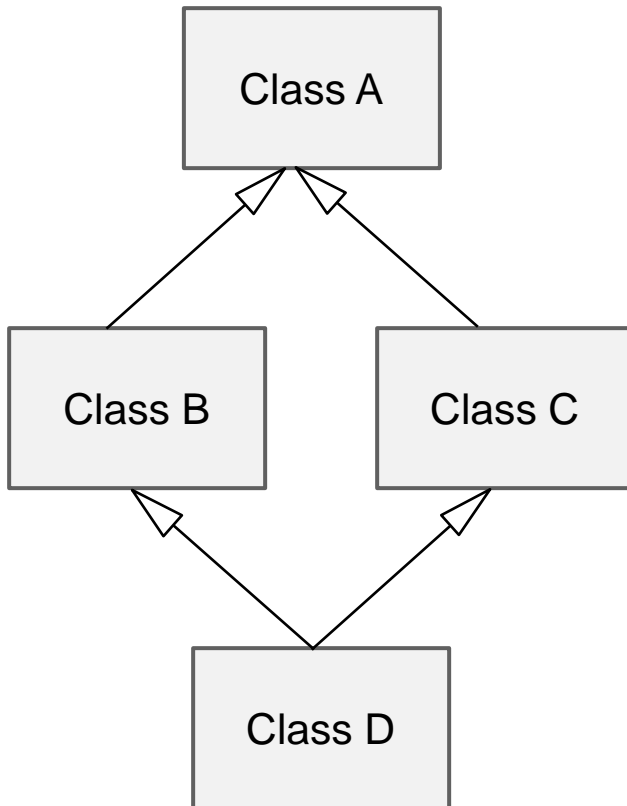
```
class C : public B {
public:
    C() {
        cout << " C's constructor is executed"
              << endl;
    }
    ~C() {
        cout << " C's destructor is executed"
              << endl;
    }
};

int main() {
    C c;
    return 0;
}
```

Output:

```
A's constructor is executed
B's constructor is executed
C's constructor is executed
C's destructor is executed
B's destructor is executed
A's destructor is executed
```

Virtual Base Classes



```
class A
class B : public A
class C : public A
class D : public B,C
```

المشكلة: المعطيات والتوابع الأعضاء في الصف A تُورث مرتين عند خلق غرض من الصف D

الحل: استخدام الصف الأساسي الافتراضي virtual base class

```
class A
class B : virtual public A
class C : virtual public A
class D : public B,C
```

An Example - Virtual Base Classes

```
class A {  
public:  
    A() {  
        cout <<"A's constructor"<< endl;  
    }  
    void f() {  
        cout <<"Test"<< endl;  
    }  
};  
class B: virtual public A {  
public:  
    B() {  
        cout <<"B's constructor"<< endl;  
    }  
};
```

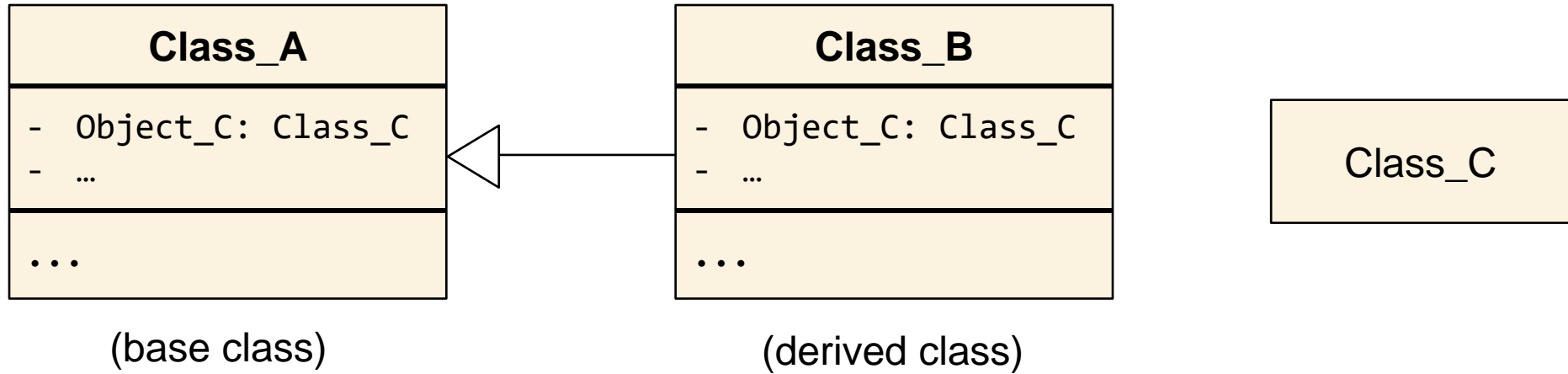
```
class C: virtual public A {  
public:  
    C() {  
        cout <<"C's constructor"<< endl;  
    }  
};  
class D: public B, C {  
public:  
    D() {  
        cout <<"D's constructor"<< endl;  
    }  
};
```

```
int main() {  
    D d;  
    d.f();  
}
```

Output:

```
A's constructor  
B's constructor  
C's constructor  
D's constructor  
Test
```


المشيّيات والهادم - أحد الأعضاء هو غرض من صف آخر-



- عندما يتم انشاء غرض من الصف **class_B**:
 1. يتم تنفيذ مشيّيات الأغراض الأعضاء في الصف **class_A**
 2. يتم تنفيذ مشيّد الصف **class_A**
 3. يتم تنفيذ مشيّيات الأغراض الأعضاء في الصف **class_B**
 4. يتم تنفيذ مشيّد الصف **class_B**
- عند تدمير (هدم) غرض من الصف **class_B**, يتم استدعاء الهوام بالترتيب العكسي لما سبق

- يتم تعريف السجلات باستخدام الكلمة المفتاحية `struct`
 - السجل هو عبارة عن صف حيث كل معطياته عامة (`public`) بشكل افتراضي
 - يمكن للسجل أن يستخدم محددات الوصول (`public`, `private`, `protected`) ويمكن أن يعرف توابع
 - يُنصح باستخدام السجل كحاوية بسيطة لمعطيات, وفي حال الحاجة إلى تابع ضمنه, يجب أن نستخدم الصف بدلاً منه
- https://google.github.io/styleguide/cppguide.html#Structs_vs._Classes

الصيغة العامة

```
struct type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    .  
    .  
} object_names;
```

```
struct Student {  
    string name;  
    int ID = 556;  
};  
  
void PrintStruct(const Student &s) {  
    cout << s.name << "  " << s.ID << endl;  
}  
  
int main() {  
    Student std1 = { "Jon Doe", 9 };  
    PrintStruct(std1);  
    PrintStruct( {"Name Test", 10});  
    return 0;  
}
```

Output:

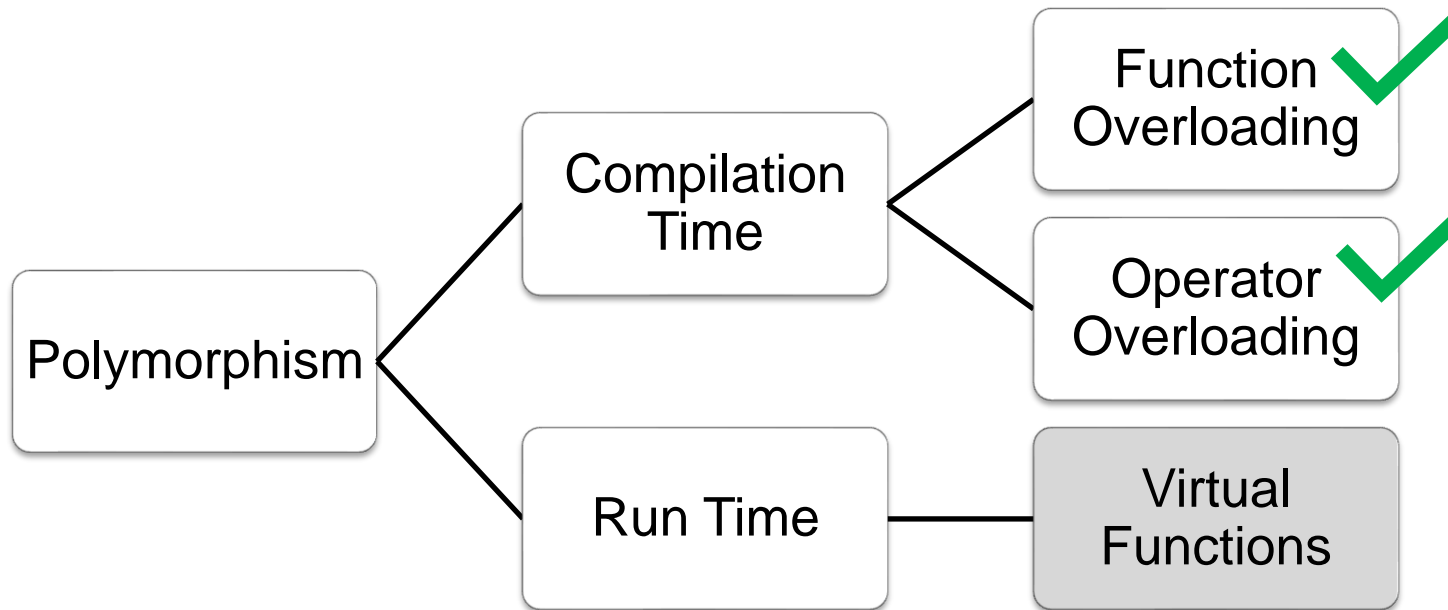
```
Jon Doe  9  
Name Test 10
```

- يمكن لسجل أن يرث من سجل أو من صف
- السجل و الصف متشابهان في الخصائص باستثناء مايلي:

	C++ class	C++ struct
Default Inheritance	private	public
Default Access Level for Member Variables and Functions	private	public

- **Polymorphism (means having many forms)**
- أحد الميزات الأساسية في الوراثة هي أنه "المؤشر إلى صف مشتق هو متوافق (type-compatible) مع نوع المؤشر إلى صفه الأساسي"
- تعدد الأشكال هي استخدام لهذه الميزة
- عند استدعاء تابع عضو, يتم تنفيذ تابع مختلف بناء على نوع الغرض الذي استدعى ذلك التابع (refer to slides 195,196)

Polymorphism Types



- virtual function هو تابع عضو في الصف الأساسي يمكن إعادة تعريفه في الصف المشتق, باستخدام الكلمة المفتاحية `virtual`
- تقوم هذه الكلمة المفتاحية بتمكين استدعاء تابع عضو في الصف المشتق لديه نفس الاسم في الصف الأساسي, عندما يكون هذا الإستدعاء من مؤشر من نمط الصف الأساسي يشير إلى غرض من الصف المشتق
- الصف الذي يقوم بالتصريح عن تابع `virtual` أو الصف الذي يرث تابع `virtual` يسمى صف متعدد الأشكال (polymorphic class)

مثال على استخدام الـ Virtual Function 2 / 1

```
class Shape {
protected:
    double width, height;
public:
    Shape(int a = 0, int b = 0) {
        width = a;
        height = b;
    }
    double getArea() {
        cout << "Area can not be calculated..."
              << "Shape must be specified first ";
        return 0;
    }
};

class Rectangle: public Shape {
public:
    Rectangle(double w, double h) :
        Shape(w, h) {
    }
    double getArea() {
        cout << "Rectangle area : ";
        return width * height;
    }
};
```

```
class Triangle: public Shape {
public:
    Triangle(double w, double h) :
        Shape(w, h) {
    }

    double getArea() {
        cout << "Triangle area : ";
        return (width * height / 2);
    }
};
```

```
int main() {
    Rectangle rect(4, 5);
    Triangle triA(4, 5);

    Shape *shape1 = &rect;
    Shape *shape2 = &triA;

    Shape *shape3 = new Rectangle(3,7);

    cout << shape1->getArea() << endl;
    cout << shape2->getArea() << endl;
    cout << shape3->getArea() << endl;
}
```

Output:

```
Area can not be calculated...Shape must be specified first 0
Area can not be calculated...Shape must be specified first 0
Area can not be calculated...Shape must be specified first 0
```


مثال على استخدام الـ Virtual Function 2 / 1

```
class Shape {
protected:
    double width, height;
public:
    Shape(int a = 0, int b = 0) {
        width = a;
        height = b;
    }
    virtual double getArea() {
        cout << "Area can not be calculated..."
              << "Shape must be specified first ";
        return 0;
    }
};

class Rectangle: public Shape {
public:
    Rectangle(double w, double h) :
        Shape(w, h) {
    }
    double getArea() {
        cout << "Rectangle area : ";
        return width * height;
    }
};
```

```
class Triangle: public Shape {
public:
    Triangle(double w, double h) :
        Shape(w, h) {
    }

    double getArea() {
        cout << "Triangle area : ";
        return (width * height / 2);
    }
};
```

```
int main() {
    Rectangle rect(4, 5);
    Triangle triA(4, 5);

    Shape *shape1 = &rect;
    Shape *shape2 = &triA;

    Shape *shape3 = new Rectangle(3,7);

    cout << shape1->getArea() << endl;
    cout << shape2->getArea() << endl;
    cout << shape3->getArea() << endl;
}
```

Output:

```
Rectangle area : 20
Triangle area : 10
Rectangle area : 21
```

Pure Virtual Function

```
class Shape {
protected:
    double width, height;
public:
    Shape(double w, double h){
        width = w;
        height = h;
    }
    virtual double getArea() = 0;
};

class Rectangle: public Shape {
public:
    Rectangle(double w, double h):Shape(w,h) { }
    double getArea() {
        cout << "Rectangle area : ";
        return width * height;
    }
};

int main() {
    Rectangle rect(4, 5);
    cout << rect.getArea();
}
```

Output:
Rectangle area : 20

تعريف تابع الافتراضي بشكل
كامل (pure virtual)

- الصف الذي يحتوي على الأقل تابع افتراضي بشكل كامل يسمى صف مجرد (Abstract class)

- لا يمكن إنشاء غرض من صف مجرد

Shape s;

error: cannot declare variable 's' to be of abstract type 'Shape'

- لكن يمكن التصريح عن مؤشر إلى نمط من صف مجرد:

```
int main() {
    Shape *s;
    Rectangle rect(4, 5);
    s = &rect;
    cout << s->getArea();
}
```

Another use of "Scope resolution operator" ::

```
class Account {
public:
    Account(string accountName);
    void setName(string accountName);
    string getName() const;
private:
    string name;
};

Account::Account(string accountName)
    : name{accountName} { }

void Account::setName(string accountName) {
    name = accountName;
}

string Account::getName() const {
    return name;
}

int main() {
    Account account1{"Jane Green"};
    Account account2{"John Blue"};
    cout << "account1 name is: " << account1.getName() << endl;
    cout << "account2 name is: " << account2.getName() << endl;
}
```

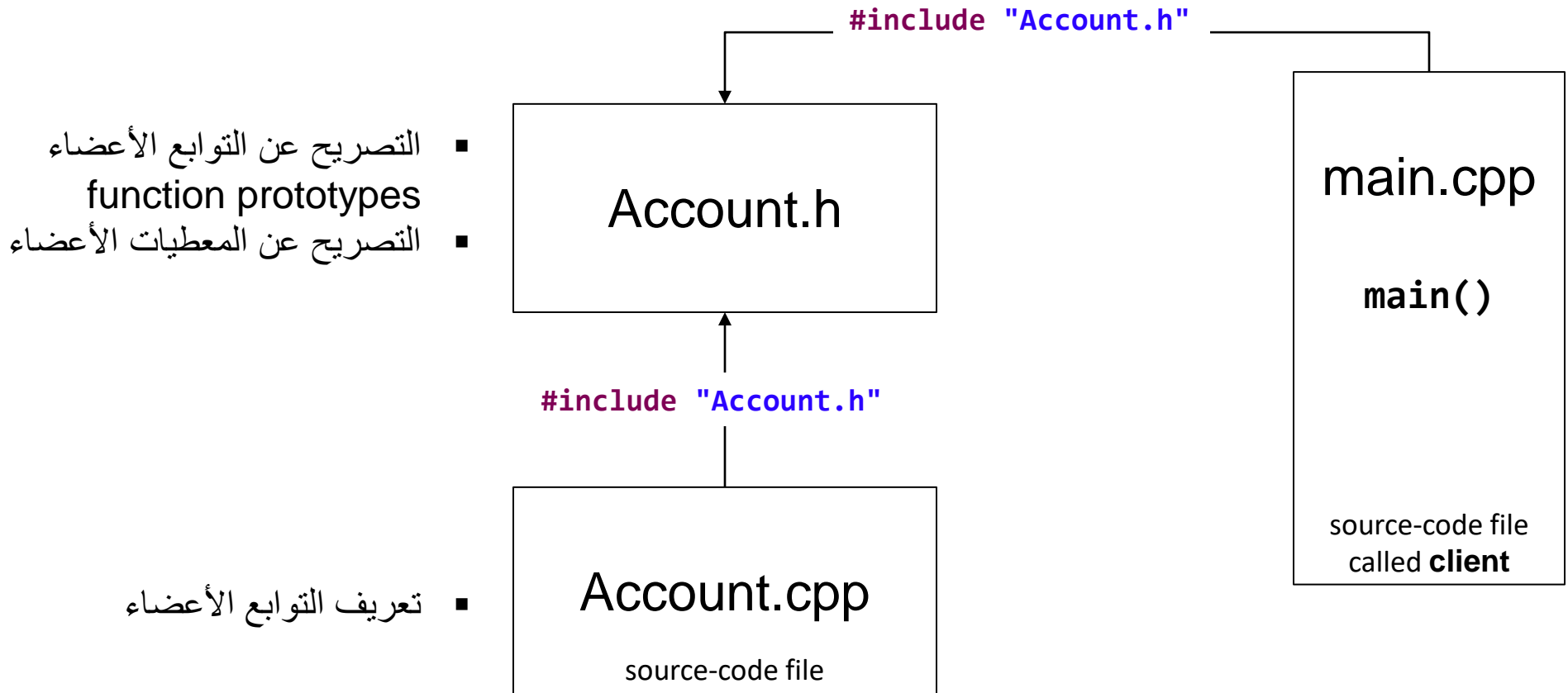
- يمكن استخدام المُعامل :: : لتعريف التوابع الأعضاء لصف خارج الصف

Output:

```
account1 name is: Jane Green1
account2 name is: John Blue
```

- فصل الواجهة (interface) عن التطبيق (implementation)
- تصف الواجهة الخدمات التي يستطيع الزبون (client) أن يستخدمها, وكيفية استدعاؤها
- لا تصف الواجهة كيف يتم تنفيذ هذه الخدمات
- غالباً يتم إنشاء وترجمة واجهة (interface) وتطبيق (implementation) للصف من قبل أحد المبرمجين (class-implementation programmer), ويتم استخدامها من مبرمج آخر (client-code programmer)

استخدام الواجهات (Interfaces) 4 / 2



استخدام الواجهات 4 / 3 (Interfaces)

```
// file's name: Account.h
#include <string>
class Account {
public:
    Account(std::string accountName);
    void setName(std::string accountName);
    std::string getName() const;
private:
    std::string name;
};
```

```
// file's name: Account.cpp
#include <string>
#include "Account.h"
Account::Account(std::string accountName)
    : name{accountName} { }

void Account::setName(std::string accountName) {
    name = accountName;
}

std::string Account::getName() const {
    return name;
}
```

```
// file's name: main.cpp
#include <iostream>
#include "Account.h"
using namespace std;

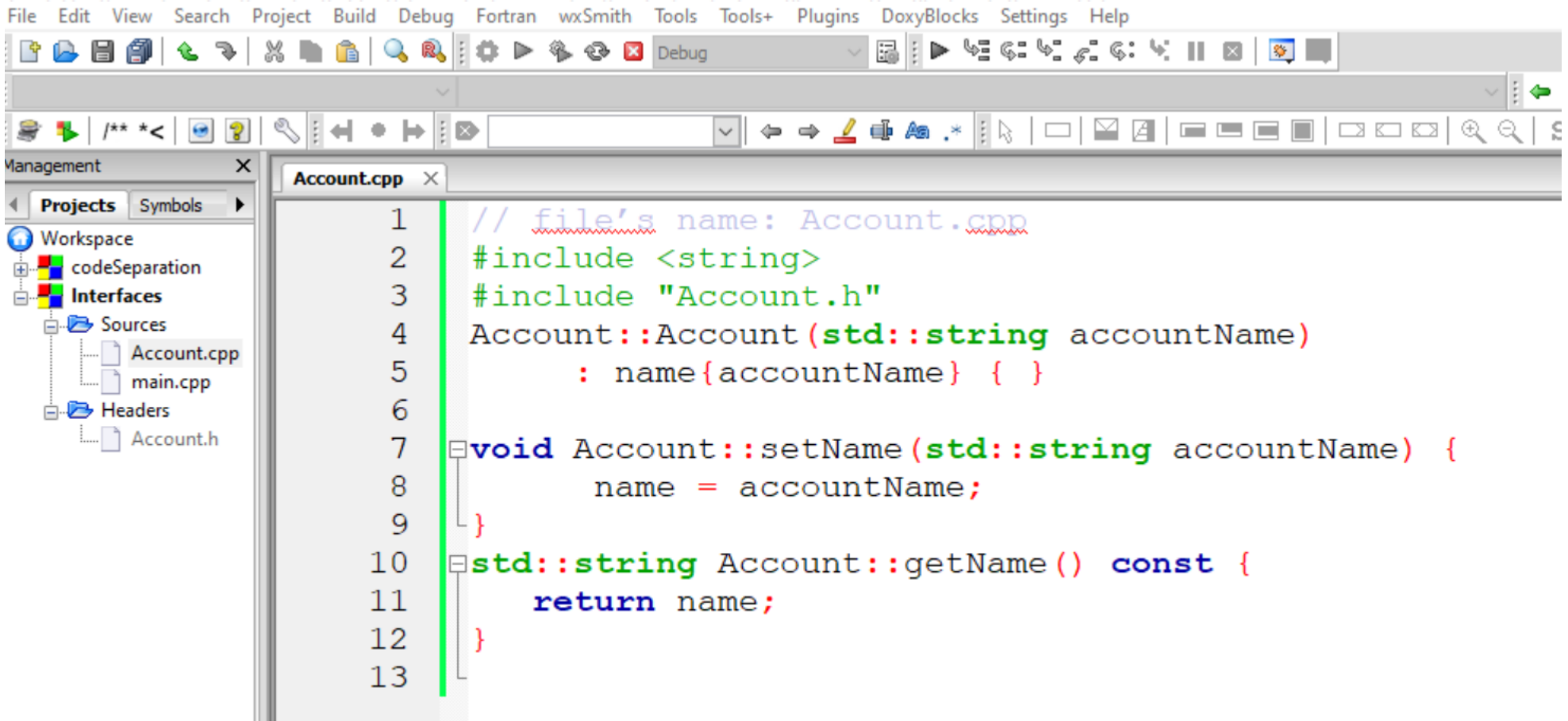
int main() {
    Account account1{"Jane Green"};
    Account account2{"John Blue"};
    cout << "account1 name is: " <<
        account1.getName() << endl;
    cout << "account2 name is: " <<
        account2.getName() << endl;
}
```

Output:

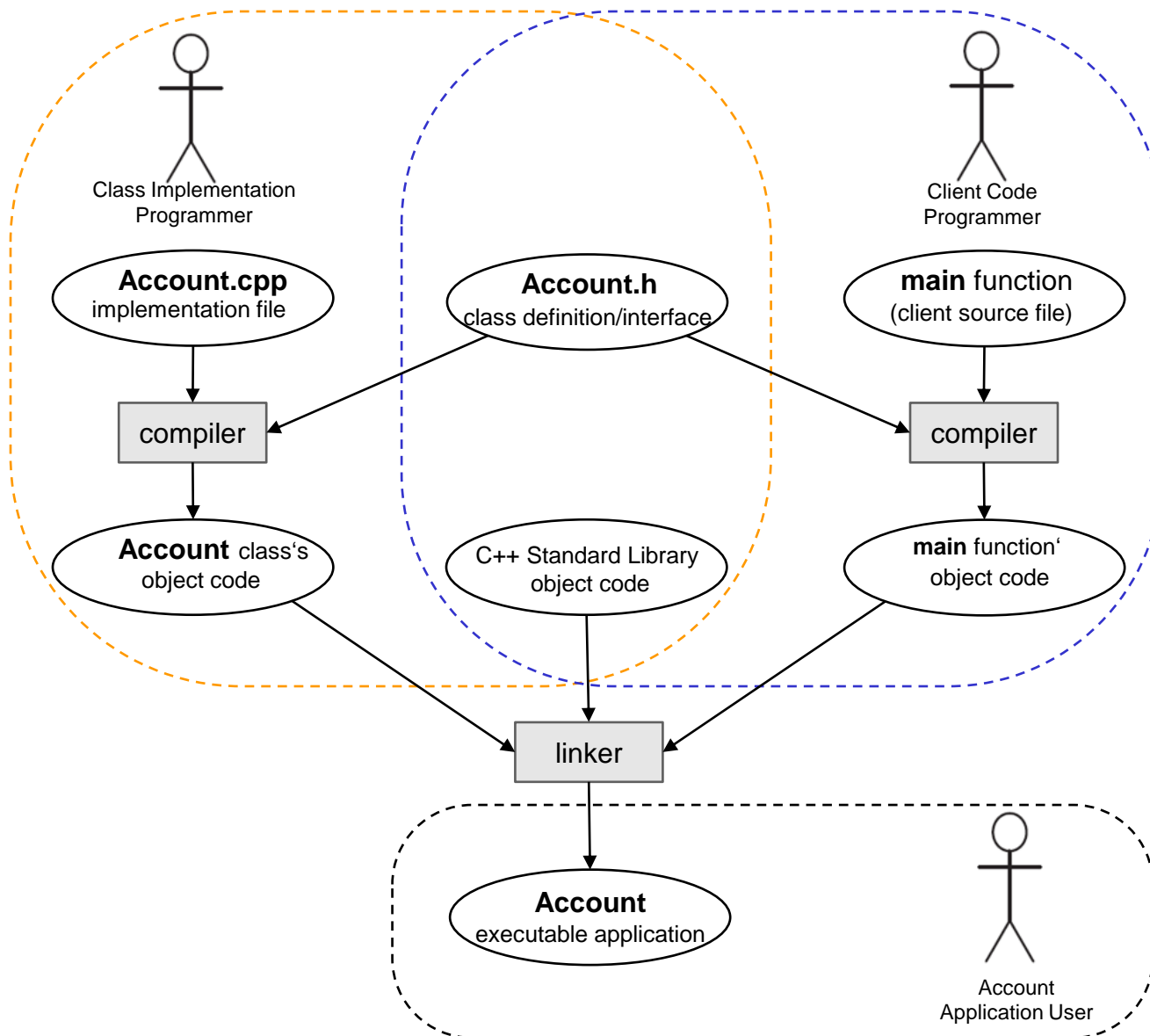
```
account1 name is: Jane Green
account2 name is: John Blue
```

استخدام الواجهات (Interfaces) 4 / 4

Account.cpp [Interfaces] - Code::Blocks 17.12



```
1 // file's name: Account.cpp
2 #include <string>
3 #include "Account.h"
4 Account::Account(std::string accountName)
5     : name{accountName} { }
6
7 void Account::setName(std::string accountName) {
8     name = accountName;
9 }
10 std::string Account::getName() const {
11     return name;
12 }
13
```



- قوالب التتابع Function Templates
- قوالب الصفوف Class Templates



<https://vvvv.org/blog/generic-nodes-project>

- يمكن استخدام قوالب التوابع لتعريف التوابع التي تُنجز عمليات متطابقة لكل نمط
– Identical logic and operation
- يتم كتابة تعريف لقالب تابع
– يمكن تسمية هذا التابع أيضاً بتابع عام generic function
- يقوم المترجم بتوليد نسخة من التابع لكل نمط يمرر كوسيط عند استدعاء التابع
– عند تمرير وسيط من النمط int يتم توليد نسخة من التابع للنمط int
– عند تمرير وسيط من النمط char يتم توليد نسخة من التابع للنمط char
- الصيغه العامة لاستدعاء التابع عندما يكون قالب (template):
`function_name <type> (parameters);`

Examples 1/3 - Function Templates قوالب التتابع

```
template <typename T> // or template< class T >
T getMax (T x, T y) {
    return (x > y) ? x : y;
}

int main() {
    cout << getMax <int> (3, 7) << endl; // Call getMax for int
    cout << getMax <double> (3.2, 7.5) << endl; // call getMax for double
    cout << getMax <char> ('m', 'y') << endl; // call getMax for char
    return 0;
}
```

Output:

7

7.5

y

ما هو الفرق بين قالب التابع وتحميل التتابع؟

Examples 2/3 - Function Templates قوالب التوابع

- التصريح عن قالب تابع بوسطاء من أنماط مختلفة

```
template <typename T, typename U> // or template< class T, class U >
T getMax(T x, U y) {
    return (x > y) ? x : y;
}
```

- يمكن أيضا تحديد وسطاء القوالب كمايلي:

```
template <typename T, int N>
T PowerToN(T x) {
    if (N ==0) return 1;
    T result = x;
    for (int i = 1; i < N; i++) {
        result *= x;
    }
    return result;
}
...
cout << PowerToN <int,5> (3);
```

حيث N قيمة ثابتة لا يمكن تغييرها

Examples 3/3 - Function Templates قوالب التوابع

```
template <class T>
void f(T x) {
    static int count = 0;
    cout << "x=" << x
         << "      count=" << count++
         << endl;
}
int main() {
    f<int>(1);
    f<int>(2);
    f<double>(1.1);
    return 0;
}
```

Output:

x=1	count=0
x=2	count=1
x=1.1	count=0

Examples 1/2 - Class Templates قوالب الصفوف

```
template <class T>
class MyPair {
    T x, y;
public:
    MyPair(T first, T second) {
        x = first;
        y = second;
    }
    T getmax();
};
```

كما قوالب التوابع, قوالب الصفوف مفيدة
عندما يعرف الصف شيء ما بشكل
مستقل عن الأنماط

```
template<class T>
T MyPair<T>::getmax() {
    T max;
    max = x > y ? x : y;
    return max;
}

int main() {
    MyPair <int> myObject(100, 75);
    cout << myObject.getmax();
    return 0;
}
```

Examples 2/2 - Class Templates قوالب الصفوف

```
template<class T>
class MyPair {
    T x, y;
    static int count;
public:
    MyPair(T first, T second) {
        x = first;
        y = second;
        count++;
    }
    static int getCount() {
        return count;
    }
    T getMax() {
        T max;
        max = x > y ? x : y;
        return max;
    }
};
```

```
template<class T>
int MyPair<T>::count = 0;

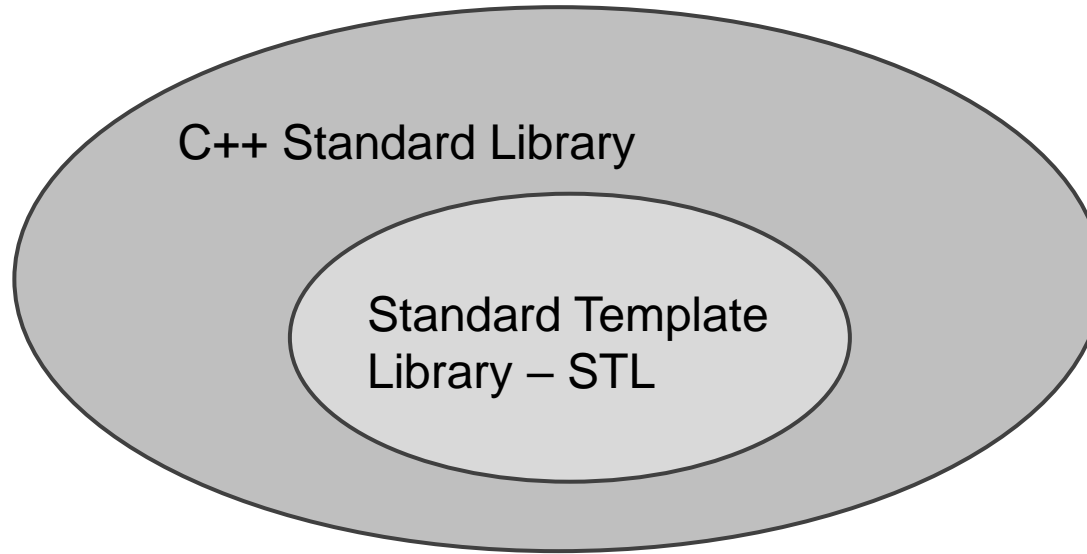
int main() {
    MyPair<int> myPair1(100, 75);
    MyPair<double> myPair2(100, 75);
    cout << myPair1.getMax() << endl;
    cout << MyPair<int>::getCount() << endl;
    return 0;
}
```

Output:

100

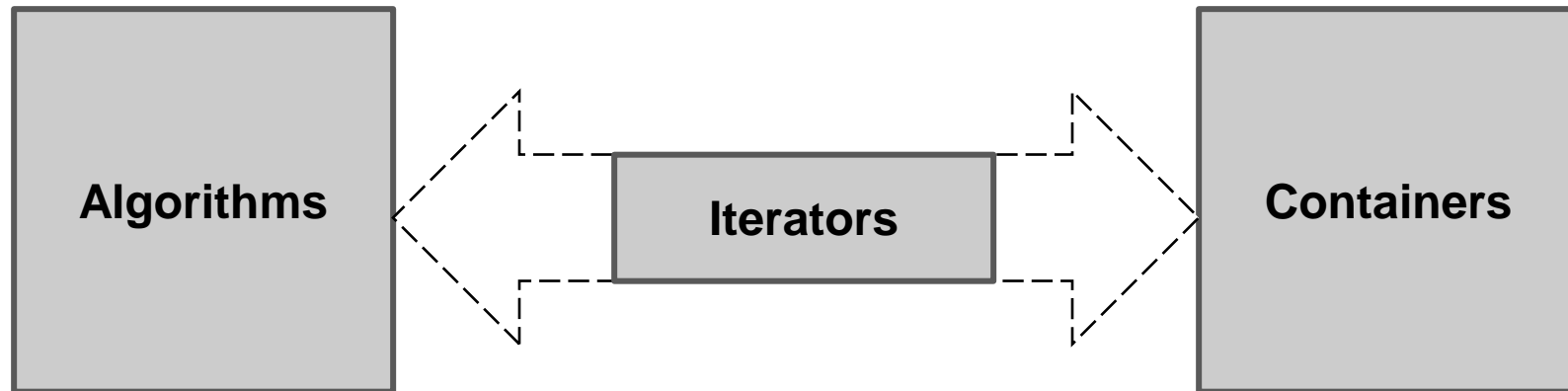
1

مقدمة في مكتبة القوالب القياسية (Standard Template Library – STL)



<https://datastructures.maximal.io/stl/>

- مكتبة القوالب القياسية STL هي جزء من مكتبة C++ القياسية, وهي مكتبة عامة تقدم حلول للتعامل مع مجموعات من البيانات باستخدام خوارزميات فعّالة



- `sort()`
- `binary_search()`
- ...

- `array`
- `vector`
- ...

- تعمل الخوارزميات (Algorithms) على عناصر الحاويات (Containers Elements) بواسطة كائنات التكرار Iterators

- array هي قالب صف (class template) من قوالب صفوف المكتبة القياسية STL

```
template < class T, size_t N > class array
```

- لاستخدام قالب الصف array `#include <array>`

- للتصريح عن مصفوفة:

```
array< type, arraySize > arrayName;
```

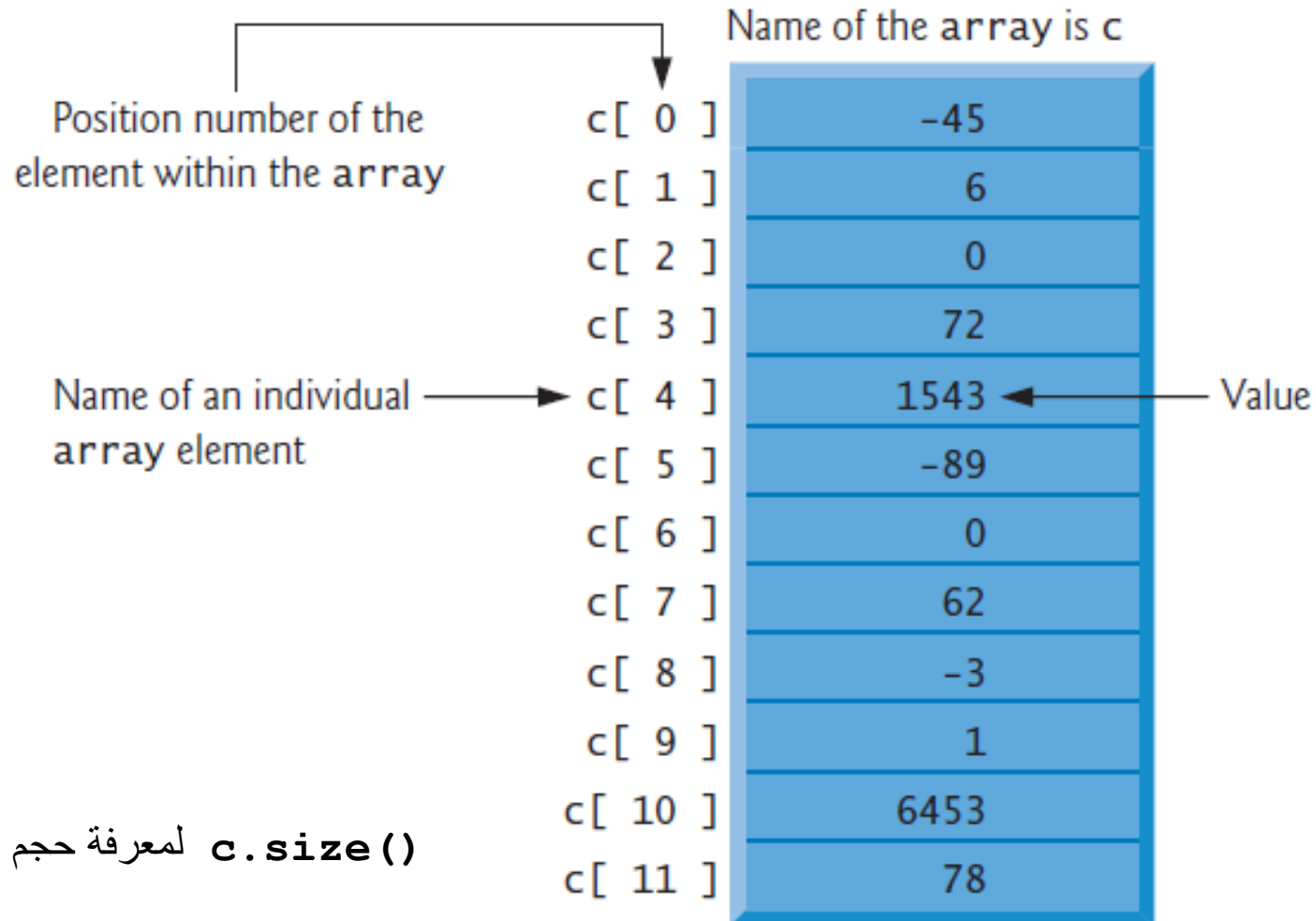
- يقوم المترجم بحجز الذاكرة المناسبة لمصفوفة بالإعتماد على **نمط (type)** عناصرها و**حجمها (array size)**

- حجم المصفوفة ثابت و يجب أن يكون unsigned int

- التصريح التالي يُعلم المترجم لحجز 12 عنصر من النمط int:

```
array< int, 12 > c; // c is an array of 12 int values
```

`array< int, 12 > c`



`c.size()` لمعرفة حجم المصفوفة

```
int main() {  
    array< int, 5 > n; // n is an array of 5 int values  
  
    // initialize elements of array n to 0  
    for (size_t i = 0; i < n.size(); ++i)  
        n[i] = 0; // set element at location i to 0  
  
    cout << "Element" << setw(13) << "Value" << endl;  
  
    // output each array element's value  
    for (size_t j = 0; j < n.size(); ++j)  
        cout << setw(7) << j << setw(13) << n[j] << endl;  
}
```

Output:

Element	Value
0	0
1	0
2	0
3	0
4	0

تهيئة عناصر مصفوفة أثناء التصريح عنها

```
int main() {  
    // use list initializer to initialize array n  
    array< int, 5 > n = { 32, 27, 64, 18, 95 };  
  
    cout << "Element" << setw(13) << "Value" << endl;  
  
    // output each array element's value  
    for (size_t i = 0; i < n.size(); ++i)  
        cout << setw(7) << i << setw(13) << n[i] << endl;  
}
```

Output:

Element	Value
0	32
1	27
2	64
3	18
4	95

```
// Set array s to the even integers from 2 to 10.

int main() {
    // constant variable can be used to specify array size
    const size_t arraySize = 5; // must initialize in declaration

    array<int, arraySize> s; // array s has 5 elements

    for (size_t i = 0; i < s.size(); ++i) // set the values
        s[i] = 2 + 2 * i;

    cout << "Element" << setw(13) << "Value" << endl;

    // output contents of array s in tabular format
    for (size_t j = 0; j < s.size(); ++j)
        cout << setw(7) << j << setw(13) << s[j] << endl;
}
```

Output: Element	Value
0	2
1	4
2	6
3	8
4	10

اختبار حدود مصفوفة (bounds checking)

- لا تقدم C++ اختبار تلقائي لحدود مصفوفة, وبالتالي لا تمنع من الإشارة إلى عنصر غير موجود
- الإشارة إلى عنصر خارج حدود المصفوفة هو خطأ منطقي (logical error) خلال زمن تنفيذ البرنامج
- خلال الوصول إلى عناصر مصفوفة عن طريق الحلقات التكرارية يجب الإنتباه إلى أن الـ index لا يأخذ قيمة أصغر من 0 أو قيمة أكبر أو تساوي العدد الكلي لعناصر مصفوفة (حجم المصفوفة)
- الكتابة لعنصر خارج حدود مصفوفة يمكن أن تفسد (corrupt) بيانات البرنامج في الذاكرة
- قالب الصف array يملك التابع at لإختبار حدود المصفوفة

- يعيد التابع `at()` عنصر المصفوفة الموجود في الموقع `i`:
`array_name.at(i)`
- يختبر هذا التابع فيما إذا كان الموقع `i` هو موقع صحيح (`valid`)

```
int main() {
    array<int, 10> myArray;

    // assign some values:
    for (int i = 0; i < 10; i++)
        myArray.at(i) = i + 1;

    // print content:
    cout << "myarray contains:";
    for (int i = 0; i < 10; i++)
        cout << " " << myArray.at(i);

    return 0;
}
```

Block scope!

```
try {  
    statements  
    // code that can throw an exception  
    // An exception can be thrown explicitly  
}  
  
catch (one parameter)  
{  
    statements  
    // code that handles the thrown exception  
}
```

Block scope!

- يمكن رمي (throw) الأنماط الأساسية و الأغراض كإستثناءات (exceptions)

```
int main() {  
    array<int, 10> myArray;  
    for (int i = 0; i < 10; i++)  
        myArray.at(i) = i + 1;  
    try{  
        cout << "myarray contains:";  
        for (int i = 0; i <= 10; i++)  
            cout << " " << myArray.at(i);  
    }  
    catch (out_of_range &e){  
        cout << "An exception occurred." << e.what() << '\n';  
    }  
    return 0;  
}
```

Output:

myarray contains: 1 2 3 4 5 6 7 8 9 10 An exception occurred.array::at: __n
(which is 10) >= _Nm (which is 10)

catch , throw , try

```
int main() {  
    int x = -1;  
    cout << "Before try \n";  
    try {  
        cout << "Inside try \n";  
        if (x < 0) {  
            throw x;  
            cout << "After throw \n";  
        }  
    } catch (int x) {  
        cout << "Exception Caught \n";  
    }  
    cout << "After catch \n";  
    return 0;  
}
```

Output:

Before try
Inside try
Exception Caught
After catch

يمكن معالجة أي

استثناء باستخدام

catch (...)

```
int main() {  
    try {  
        throw 10;  
    } catch (char *excp) {  
        cout << "Caught " << excp;  
    } catch (...) {  
        cout << "Default Exception\n";  
    }  
    return 0;  
}
```

Output:

Default Exception

```
int main() {  
    try {  
        throw 'a';  
    } catch (int x) {  
        cout << "Caught ";  
    }  
    return 0;  
}
```

the program terminates abnormally!

تعليلة for المعتمدة على مجال (range-based for)

- للوصول ومعالجة عناصر مصفوفة دون الحاجة إلى عداد (counter)
- استخدام هذه التعليلة يجنب الخروج خارج حدود المصفوفة

Syntax

```
for ( range_declaration : range_expression )  
    loop_statement
```

نمط + معرف, مثال: int item

اسم المصفوفة

- النمط في range_declaration يجب أن يكون نفس نمط عناصر المصفوفة
- يمكن استخدامها مع المصفوفات المدمجة (built-in arrays) أيضاً

```
int main() {  
    array< int, 5 > items = { 1, 2, 3, 4, 5 };  
  
    // display items before modification  
    cout << "items before modification: ";  
    for (int item : items)  
        cout << item << " ";  
  
    // multiply the elements of items by 2  
    cout << "\nitems after modification: ";  
    for (int &itemRef : items) ←  
        itemRef *= 2;  
  
    for (int item : items)  
        cout << item << " ";  
}
```

يجب استخدام المرجع في حال أردنا تغيير قيم عناصر مصفوفة

Output:

```
items before modification: 1 2 3 4 5  
items after modification: 2 4 6 8 10
```

! لا يمكن الوصول إلى الـ index عند استخدام range-based for

ترتيب مصفوفة sort an array

- لترتيب مصفوفة يمكن مثلا استخدام التابعين: يجب تضمين الملف الرئيسي `#include <algorithm>`
 - sort لترتيب المصفوفة تصاعديا ascending --- function template!
 - reverse لترتيب المصفوفة تنازليا descending --- function template!

```
int main() {  
    array<int, 5> myArray = { 2, 35, 20, 10, 100 };  
  
    cout << "myArray in ascending order : ";  
    sort(myArray.begin(), myArray.end());  
    for (int item : myArray)  
        cout << " " << item;  
    cout << endl;  
  
    cout << "myArray in descending order : ";  
    reverse(myArray.begin(), myArray.end());  
    for (int item : myArray)  
        cout << " " << item;  
    cout << endl;  
}
```

- يجب تضمين الملف الرئيسي

`#include <algorithm>`

- **begin()** This function is used to return the **beginning position** of the container.
- **end()** This function is used to return the **after end position** of the container.

البحث ضمن مصفوفة searching within an array

- يمكن استخدام التابع `binary_search` للبحث عن عنصر ضمن مصفوفة --- function template!

- هذا التابع يعمل فقط في حال المصفوفة مرتبة

```
int main() {
    array<int, 5> myArray = { 2, 35, 20, 10, 100 };

    sort(myArray.begin(), myArray.end());
    cout << "myArray in ascending order : ";
    for (int item : myArray)
        cout << " " << item;
    cout << endl;

    int x;
    cin >> x;
    // This searching only works when container is sorted!!
    bool found = binary_search(myArray.begin(), myArray.end(), x);
    cout << x << (found ? "was" : "was not")
        << " found in myArray" << endl;
}
```


المصفوفات متعددة الأبعاد multidimensional arrays

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Diagram illustrating a 2D array structure with 3 rows and 4 columns. The array is named 'a'. The row subscript is the first index, and the column subscript is the second index. The array name 'a' is indicated by an arrow pointing to the first subscript.

- مصفوفة مؤلفة من 3 أسطر و 4 أعمدة

- كل عنصر في مصفوفة a يمثل بالشكل $a[i][j]$

- التصريح عن المصفوفة a كقالب مصفوفة:

number of columns

```
array< array< int, 4 >, 3 > a;
```

number of rows

استخدام "nested range-based for" مع المصفوفات متعددة الأبعاد

```
const size_t rows = 2;
const size_t columns = 3;
void printArray(const array<array<int, columns>, rows> &);

int main() {
    array<array<int, columns>, rows> array1 = { 1, 2, 3, 4, 5, 6 };
    static array<array<int, columns>, rows> array2;
    cout << "Printing array1" << endl;
    printArray(array1);

    cout << "Printing array2" << endl;
    printArray(array2);
}

void printArray(const array<array<int, columns>, rows> &a) {
    // loop through array's rows
    for (auto const &row : a) {
        // loop through columns of current row
        for (auto const &element : row)
            cout << element << " ";
        cout << endl;
    } // end outer for
}
```

Output:

```
Printing array1
1 2 3
4 5 6
Printing array2
0 0 0
0 0 0
```

- vector هو قالب صف (class template) من قوالب صفوف المكتبة القياسية STL

```
template <
```

```
class T, class Allocator = std::allocator<T>
```

```
> class vector
```

- قالب الصف vector شبيه بقالب الصف array, ولكن يمكن تغيير حجمه خلال تنفيذ البرنامج

```
#include <vector> لاستخدام قالب الصف vector
```

- للتصريح عن vector:

```
vector< type > vectorName;
```

- التصريح عن vector الذي يحوي أعداد صحيحة:

```
vector< int > integers;
```

- يمكن التصريح عن vector وتهيئته بحجم: -- يتم تهيئة كل عنصر بقيمة 0

```
vector< int > integers (5);
```

- يمكن التصريح عن vector وتهيئته بحجم بالإضافة إلى تهيئة عناصره بقيمة:

```
vector< int > integers (5,3);
```

- يمكن التصريح عن vector وتهيئة عناصره بقيم محددة لكل عنصر:

```
vector< int > integers{1,2,5,5,3};
```

counter-based for

```
int main() {  
    // Declare a vector of ten numbers  
    vector<double> vec(10);  
    cout << "Please enter 10 numbers: ";  
  
    for (size_t i=0; i<10; i++)  
        cin >> vec[i];  
    // Print the vector's contents  
    for (size_t i=0; i<10; i++)  
        cout << vec[i] << '\n';  
}
```

range-based for

```
int main() {  
    // Declare a vector of ten numbers  
    vector<double> vec(10);  
    cout << "Please enter 10 numbers: ";  
  
    for (double &elem : vec)  
        cin >> elem;  
    // Print the vector's contents  
    for (double elem : vec)  
        cout << elem << '\n';  
}
```

توابع يمكن أن تستخدم مع vectors

<code>push_back(x)</code>	إضافة عنصر في آخر الـ vector
<code>pop_back()</code>	حذف آخر عنصر من الـ vector
<code>insert(position, x)</code>	إضافة عنصر إلى الـ vector قبل موقع محدد
<code>operator[]</code>	الوصول إلى عنصر من الـ vector
<code>at(i)</code>	الوصول إلى العنصر بالموقع i مع اختبار حدود الـ vector
<code>size()</code>	عدد العناصر الحالية في الـ vector
<code>empty()</code>	يعيد true في حال الـ vector لا يحتوي أي عنصر
<code>clear()</code>	حذف جميع عناصر الـ vector
<code>begin()</code>	موقع العنصر الأول في الـ vector
<code>end()</code>	الموقع التالي لموقع العنصر الأخير في الـ vector

!

multidimensional vector

```
void print(const vector<vector<double>>&);

int main() {

    vector<vector<double>> matrix;
    matrix = { { 2, 4, 5 }, { 2, 7 } };
    print(matrix);

}

void print(const vector<vector<double>> &m) {
    for (size_t row = 0; row < m.size(); row++) {
        for (size_t col = 0; col < m[row].size(); col++)
            cout << setw(5) << m[row][col];
        cout << '\n';
    }
}
```

Output:

2	4	5
2	7	

تمنياتي لكم بالتوفيق...