Advanced Programming Concepts with C++ CSI2372 – Fall 2019

Jochen Lang &
Mohamed Taleb
EECS

uOttawa

L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa

uOttawa.ca

This lecture

- C-like C++
- **Memory management in C/C++**
  - Memory allocation: static, automatic and dynamic, Ch. 6.1.1, Ch. 12
  - Allocation and de-allocation, Ch. 12.1.2
  - 2-D and N-D Arrays, Ch. 3.5
  - Pass by value, by reference, by pointer, Ch. 6.2-6.2.4
  - Passing a function Ch. 10.3
    - Function pointers, Ch. 6.7

Java Memory Management

- **Java**
  - Memory is requested by the program
    - all Java objects are on the heap allocated with new
    - Local variables of primitive types and references are on the stack
  - Memory is freed by the Java garbage collector
    - After the last reference is deleted
      - garbage collector must keep track (e.g., reference counting or mark and sweep (delete))
    - When the garbage collector executes

Java Garbage Collection

- **Mark and sweep algorithm**
  - Two pass algorithm
    - Mark all objects which are reachable
    - Sweep (delete) objects which are unmarked
  - Heap is compacted
  - Sun Java JDK 1.0 and 1.1
- **Generational collector**
  - Young (“Eden” and 2 survivor), tenured and permanent spaces
  - Copying between spaces, i.e., compaction
  - Garbage collections: Minor (young space only) and major
  - Improves performance dramatically when most objects are short-lived, e.g., temporary objects
  - Sun Java JDK 1.2 and later

Advantages and Disadvantages of Garbage Collection (GC)

- **Advantages**
  - Frees program designer from thinking about GC
    - increased productivity
  - Avoids most memory bugs and leaks which can be extremely difficult to find and fix
  - Security
- **Disadvantages**
  - Overhead
    - JVM needs to run a GC thread
    - JVM needs extra memory to store extra object information
  - less control, soft real-time applications
  - often leads to memory issues being overlooked (references not being released)

User Control of Garbage Collection in Java

- **Java has minimal user control**
  - Parameters to the Java VM
  - Nulling of object references
  - Pooling of objects
  - Explicitly request a GC
  - *But still need to make sure that references are not kept too long (or forever)*
- **Results**
  - hard to predict and often counter-productive

Memory Management and Garbage Collection in C++

  - Different mechanisms depending on type of memory
  - Memory on the heap: No garbage collection!
- **Changes in C++11**
  - garbage collection interface is defined for compiler implementers
  - optional vendor-specific implementations
  - Uses notion of safe derivation of pointers but users remains in control through declaring, proposed options: relaxed (same as before), preferred, strict

Memory Allocation

- **Static Memory Allocation**
  - Allocated by the “linker” at the beginning of the program, and de-allocated when the program finishes
- **Automatic Memory Allocation**
  - Automatically allocated and de-allocated during the program execution
    - Examples include function arguments, function return values and local variables
- **Dynamic Memory Allocation**
  - Handled by explicit statements in the program. Allocation and de-allocation only by request
  - No Garbage Collection

Memory Locations

- **In the executable program code**
  - allocated when program is loaded
    - global variables, static variables
- **On the stack of the program**
  - allocated automatically during execution
    - local variables, functions parameters, functions return values
- **On the heap of the program**
  - allocated as coded during execution
    - in C++: `new new[] delete delete[]`
    - in C: `malloc calloc realloc free`

new and delete Operator/Keyword

- **`new` type and `new` type`[]`**
  - Allocates memory for objects, arrays, data types on the heap from the free store
  - Returns nonzero pointer to a memory location on success
  - May raise an exception `std::bad_alloc`
- **`delete` pointer and `delete[]` pointer**
  - De-allocates a block of memory pointed to by a pointer and returns the memory to the free store
  - Results unpredictable if used on not properly allocated memory

`new` and `delete` Example

- **`new` and `delete`**
  - used for object allocation and de-allocation
    - new is similar to Java
    - Error to call delete on variables not allocated with new
- **Example: array allocation and de-allocation**

```
const int max = 1000;
int numbers[max];
int *dynNumbers;
int arraySize = 0; cin >> arraySize;
if ( !cin.fail() ) {
   dynNumbers = new int[arraySize];
   delete[] dynNumbers;
}
delete[] numbers;   Illegal!
```

A closer Look at new and delete

- **3 steps during new**
  - allocate memory large enough to hold data of the requested type
  - construct the type
  - return a pointer to the constructed type
- **2 steps during delete**
  - destruct the type
  - return the memory to the free-store

constructor in Java and C++

finalize method in Java; destructor in C++

C-Style Memory (De-)Allocation

- No constructors and destructors in C
  - they are not called during `free` and `malloc`
- `malloc` return a pointer of `type void *`
  - memory is not initialized
  - Not type-safe
- `malloc` does not raise an exception on failure
  - Program needs to check for null pointer
- `free` behaves similar to delete
  - memory pointed to must have been allocated with `malloc`
  - multiple free calls will have unpredictable results
  - `free` with a null pointer should be Ok (ISO C)

Same example : in C and in C++

```c
char *char1, *char40;
//Allocation of one char
char1 = (char*) malloc(1);
//Allocation of a string of 40 char
char40 = (char*) malloc(40);

double *double1, *double50;
//Allocation of one double
double1 = (double*) malloc(sizeof(double));
double50 = (double*) malloc(50 * sizeof(double));
struct Node{
                int value;
                Node * left;
                Node * right;
};
Node *node1, *nodeN;
//Allocation of one node
node1 = (Node*) malloc(sizeof(Node));
//Allocation of N nodes
int N;
printf("How many nodes : “);
scanf ("%d ", &N);
nodeN = (Node*) malloc(N * sizeof(Node));
```

```cpp
char *char1, *char40;
//Allocation of one char
char1 = new char;
//Allocation of a string of 40 char
char40 = new char[40];

double *double1, *double50;
//Allocation of one double
double1 = new double;
double50 = new double(50);
struct Node{
                int value;
                Node * left;
                Node * right;
};
Node *node1, *nodeN;
//Allocation of one node
node1 = new Node;
//Allocation of N nodes
int N;
cout << “How many nodes : “;
cin >> N;
nodeN = new Node(N);
```

Array of Arrays: Definition and Initialization

- array holds garbage unless initialized
- inner braces are optional
- partial initialization is possible
- first dimension is optional
- "no" limit on the number of dimensions

```
const int numRows = 3, numCols = 4;
int numbersA[numRows][numCols]; // not initialized
int numbersB[numRows][numCols]{{ 0, 1, 2, 3 },
                               { 4, 5, 6, 7 },
                               { 8, 9, 10, 11 }};
int numbersC[numRows][numCols] { 0, 1, 2, 3, 4, 5, 6,
                                 7, 8, 9, 10, 11 };
int numbersD[][numCols]{ 0, 1, 2, 3, 4, 5, 6,
                         7, 8, 9, 10, 11 };
```

Array of Arrays

- **C/C++ does not have multidimensional arrays!**
- **BUT arrays of arrays**
  - Example:
    - array of size 3 which holds arrays of size 4

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}

a_0	a_{00}	a_{01}	a_{02}	a_{03}
a_1	a_{10}	a_{11}	a_{12}	a_{13}
a_2	a_{20}	a_{21}	a_{22}	a_{23}

Element Access and Pointers

- **Access by indices**
- **Pointer manipulation**
  - Pointer to row (array)
    - A row is a one-dimensional array, i.e., pointer to an array
  - Pointer to element
  - Pointer to first element

```
int numbers3D[3][4][5];
int element = numbers3D[1][2][0];
```

```
int numbers[3][4];
```

```
int (*row)[4] = &numbers[1];
```

```
int elementA = (*row)[3];
int* elementB = &numbers[0][2];
```

```
int* elementC = &numbers[0][0];
int* elementD = numbers[0];
```

Arrays cannot be assigned; they are not automatically copied

Semi-Dynamic Allocation of Array of Arrays

- **Only the first dimension can be determined at run-time**
  - Example: 2D array
    - number of rows at run-time
    - number of cols at compile

```
int numRows = 3;
int (*numbers)[4] = new int[numRows][4];
delete[] numbers;
```

Dynamic Allocation of Array of Arrays

- **Need to (de-)allocate all the arrays (arrays of arrays)**

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

```
for(int i=0; i<3; i++) {
  delete[] numbers[i];
}
delete[] numbers;
```

  – row + 1 call to new
    - row + 1 separate memory location may be returned
- **Contiguous memory layout may be important!**

Memory Layout of Array of Arrays

- **Pointer manipulation (C-style)**

```
int *tmp = new int[12];
int **numbers = new int*[3];
for(int r=0; r<numRows; r++) {
  numbers[r] = &tmp[r*numCols];
}
```

- **Deallocation? Ugly!**
- **Instead:**
  - Write a matrix, image etc. class with operators and accessors and use a one-dimensional array.
  - Use `std::array` (a class which wraps low-level arrays)
  - Use `std::vector` (similar to `java.util.ArrayList`)

Initialization by Value and by Pointer

- **Variable is copied**
  - Pointer is just another type

```
int *ptrNumA  =   1
                0x0100

ptrNumA  = ptrNumB = 0x0100
```

```
int numA = 1;
int numB = numA;
```

```
int *ptrNumA = &numA;
int *ptrNumB = ptrNumA;
```

- **Pass by pointer is identical to pass by value**
  - pointer is copied
  - object pointed to is not affected
    - similar effect than pass by reference as in Java
- **Note: Arrays are passed by pointer even if we use array syntax!**

Variable Initialization by Reference

- **Reference to the variable is created**
  - Same as in Java
  - Reference is an alias to a variable
    - "just another name"

```
int numA = 1;
int &numB = numA; // int * const numB = &numA;

numB =3 + numB; //(*numB) = 3 + (*numB);
numB++; // (*numB)++;
```

- **In general: Variable initialization is the same as argument passing into methods and functions: "Pass by Reference"**

```
void myFunction( int (&arg)[4] );
…
int numbers[4];
myFunction( numbers );
```

Passing Arrays

- **Low-level arrays can be passed by pointer to the first element, by pointer or by reference**

```
int a[]{0,1,2,3,4,5,6,7};
```

```
int sumOfP1( int *ptr);
int sumOfP2( int ptr[]);
int sumOfP3( int ptr[8]);
```

Pass by pointer to first element!
Size unknown!

```
for ( auto  a : ptr ) {}
```

```
int sumOfPArray( int (*ptr)[8]);
```

```
int sumOfRefArray( int (&ref)[8]);
```

Pointer (top) or reference (bottom) to an int array of size 8.

```
for ( auto  a : ref ) {}
```

Passing Arrays

- **Low-level arrays can be passed by pointer to the first element, by pointer or by reference**

```
int a[]{0,1,2,3,4,5,6,7};
```

```
int sumOfP1( int *ptr);
int sumOfP2( int ptr[]);
int sumOfP3( int ptr[8]);
```

Pass by pointer to first element!
Size unknown!

```
for ( auto  a : ptr ) {}
```

```
int sumOfPArray( int (*ptr)[8]);
```

```
int sumOfRefArray( int (&ref)[8]);
```

Pointer (top) or reference (bottom) to an int array of size 8.

```
for ( auto  a : ref ) {}
```

Function Pointers

- **STL, GUIs, etc. expect to pass a callback function.**
  - We can use objects with operator overloading (functors).
  - But in simple cases a function is enough
  - Example: lessThan

```
// Function declaration
bool lessThan(const Point2D&, const Point2D& );
// Function accepting a function pointer
const Point2D& compare( const Point2D&, const Point2D&,
                 bool (*) (const Point2D&, const Point2D&) );
// Function definition
bool lessThan(const Point2D& ptA, const Point2D& ptB ) {
  return ptA.d_components[1] < ptB.d_components[1];
}
```

Using Function Pointers

- **Function pointer types have to match**
  - arguments and *return type (unlike function overloading)*
- **Function pointers have awkward syntax**
  - We can use simplified notations, or, we can use typedefs
  - Example: lessThan (continued)

```
// Still the same function accepting a function pointer
// but no (*)
const Point2D& compare( const Point2D&, const Point2D&,
                  bool (const Point2&, const Point2D&) );
// using a typedef - pre C++11 style
typedef bool (*pt_compare)(const Point2D&, const Point2D&);
const Point2D& compare( const Point2D&, const Point2D&,
                          pt_compare );
```

Calling Function through Pointers

- explicitly dereferenced
- implicitly dereferenced

optional

```
// Function declaration
bool lessThan(const Point2D&, const Point2D& );
// no typedef
Point2D ptA, ptB;
lessThan( ptA, ptB ); // direct call of function, no ptr
bool (*ptr) ( const Point2D&, const Point2D& ) = &lessThan;
(*ptr)(ptA,ptB);
ptr(ptA,ptB);
// using a typedef C++11 notation
using pt_compare=bool (*)(const Point2D&, const Point2D&);
pt_compare c = &lessThan;
(*c)(ptA,ptB);
c(ptA,ptB);
```

References with `auto`

- **`auto` type deduction will use the base type**
  - reference is not part of base type
  - aside: top-level `const` are also not part of the base type

```
int i, &iRef = i, *iPtr = &i;
auto j = iRef; // j is not a reference
auto k = iPtr; // k is a pointer
auto &jRef = j; // jRef is a reference
```

  - But there is also `decltype` which works differently
    - Need to discuss LValue and RValue

LValue and Rvalue Expression

- **Origin of term comes simply from the place of an expression in C**

Invalid lvalue

```
int i=5;
int j = 2*i;
2*i = j;
```

Valid rvalue

- **A non-rigorous C++ definition**
  - LValue expressions refer to a memory location
    - yields an object or function
    - uses the memory location
  - RValue are the rest (non-LValue expressions).

Some Operators require LValues

- Assignment operators need LValue as left operand (the classic)
- Address-of-operator require a LValue operand
- Dereference and subscripting yields an LValue (you can assign to it).
- Increment and decrement need a LValue operand and the prefix version yields a LValue

```
int foo();
int i=5;
int *j = &i;
int array[5];
array[3] = 3;
++i *= 3; // i = (5+1)* 3 = 18
```

References with `decltype`

decltype deduces the type of the specified expression.

auto deduces types based on values being assigned to the variable.

- **`decltype` deduction will evaluate expressions**
  - brackets can be used to define references
  - using an expression that yields a LValue (can appear on the lhs of an assignment) will make it a reference
  - top-level const and reference are used to deduce type

```
int i = 5, &iRef = i, *iPtr = &i;//iRef=5, *iPtr=5
decltype(iRef) jRef = i; // reference jRef = 5
decltype(i) iVal=4; // iVal=4 is not a reference
decltype((i)) kRef = i; // reference kRef = 5
decltype(*iPtr) lRef = i; // reference lRef = 5
```

Auto with `const` and References C++14

- As we have seen `auto` uses the underlying type, e.g., `const int` or `int&` become `int`. This is the same as normal initialization rules (templates).
- `auto` and `decltype` infer types differently
- In C++14 we can use `decltype` type inference with `auto`

```
int i = 5, &iRef = i; //iRef = 5
auto j = iRef; // j is not a reference = 5
auto &jRef = i; // jRef is a reference = 5
decltype(auto) jRef2 = iRef; // jRef2 = 5 is also a reference
decltype(auto) jRef3 = (i); // jRef3 = 5 is also a reference
```

Next Lectures

OO

- **Object-oriented design**
  - Class relationships: association, aggregation, generalization and inheritance
  - Pointer attributes
  - Copy construction and assignment
  - Polymorphism: Virtual functions, abstract classes and dynamic cast
  - Exceptions Basics
  - Inline functions, static members, constexpr