Advanced Programming Concepts with C++
CSI2372 – Fall 2017

Jochen Lang
EECS, University of Ottawa
Canada

uOttawa

L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa

uOttawa.ca

This lecture

- C-like C++
- **Data Types**
  - Arrays and pointers, Ch. 3.5
  - Old-style C-strings, Ch. 3.5.4
  - Reinterpretation casts, Ch. 4.11.3
  - Scope, Storage class and Linkage
  - Storage class modifiers , Ch. 2.4
  - Type aliasing, Ch. 2.5

Arrays

- **1-D Arrays**
  - Consecutive data of the same type, similar to a vector in math
  - Concept of array is the same in Java and C++ but memory management, features and syntax is different.
- **Example double array:**

```
const int Length = 3; double coordinate[Length];
for (int i=0; i<Length; ++i ) {
  std::cout << coordinate[i] << std::endl;
  coordinate[i] = 0.5*i;
  std::cout << coordinate[i] << std::endl;
}
```

Pointer Properties

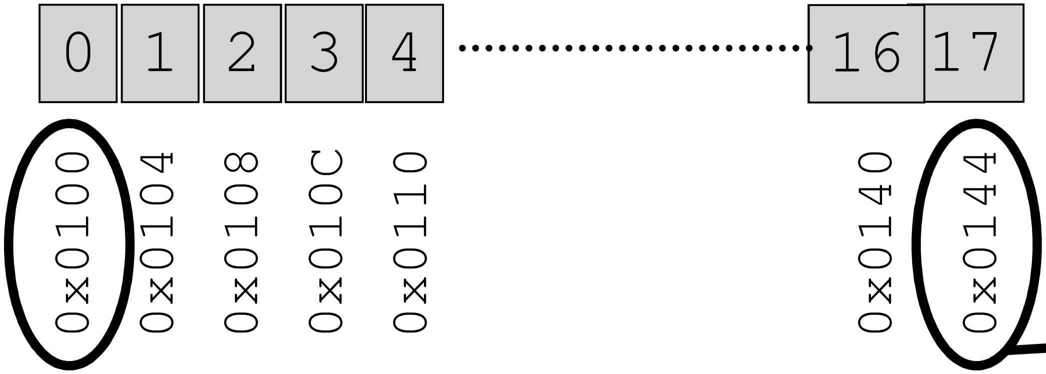
- **Pointers are iterators for arrays**
  - Pointers are fixed size and independent of data size
  - Pointers hold the address of memory
  - Operators for pointer arithmetic are

    `+   ++   -   --   =   +=   -=  ==`

```
int num[]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
              11, 12, 13, 14, 15, 16, 17 };
int *ptrNum;
ptrNum = &num[0];
cout << *ptrNum << " to " << *(ptrNum+17) << endl;
cout << ptrNum << " to " << (ptrNum+17) << endl;
cout << "Array Length (Bytes): " << (size_t)(ptrNum+17) -
 (size_t) ptrNum + 1 * sizeof(int) << endl;
```

Pointers

- **Data is located somewhere in memory**
  - Pointer “points” to a data location, it holds the address of the location



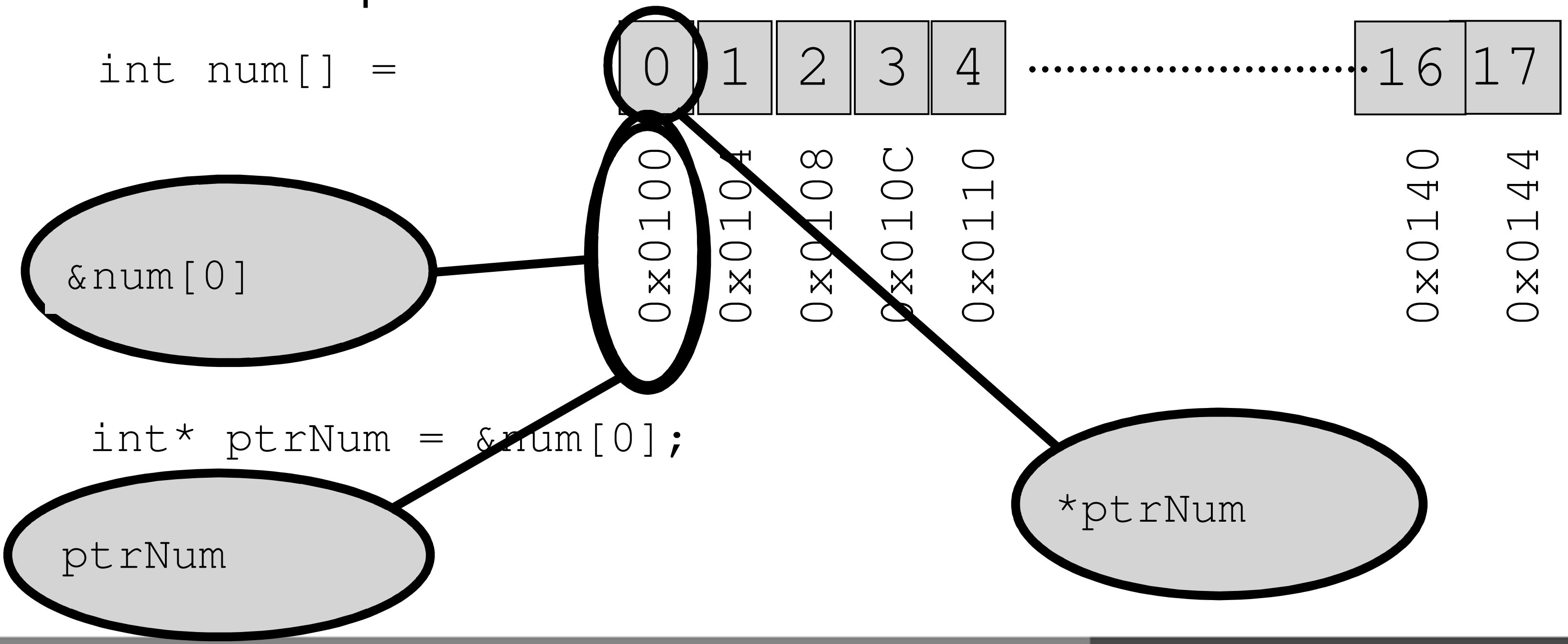
0x means hexadecimal

0x0100 + 17*4 decimal

```
int numbers[]{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                 11, 12, 13, 14, 15, 16, 17 };
int *ptrNumber = &numbers[0];
cout << *ptrNumber << " to " << *(ptrNumber+17) << endl;
cout << ptrNumber << " to " << (ptrNumber+17) << endl;
```

Address and Dereference

- **Operators**
  - Declaration of a pointer `type * pointerToType`
  - Address of operator `&`
  - Dereference operator `*`



```
int num[] =
```

```
int* ptrNum = &num[0];
```

Pointer to Array

- **Keep in mind pointers are typed!**
  - A pointer to integer is different than a pointer to integer array
  - A pointer to an integer array of size 5 is different from a pointer to integer array of size 3

```
int numbers[]{ 0, 1, 2, 3, 4};
// Access through pointer to Array
int (*ptrArray)[5] = &numbers;
cout << (*ptrArray)[3] << endl;
// Access through pointer to elements
int* firstE = &numbers[0];
cout << firstE[3] << endl;

// Mixing pointers
int (*ptrShortArray)[2] = &numbers; // Compile error
```

Old-style C-strings

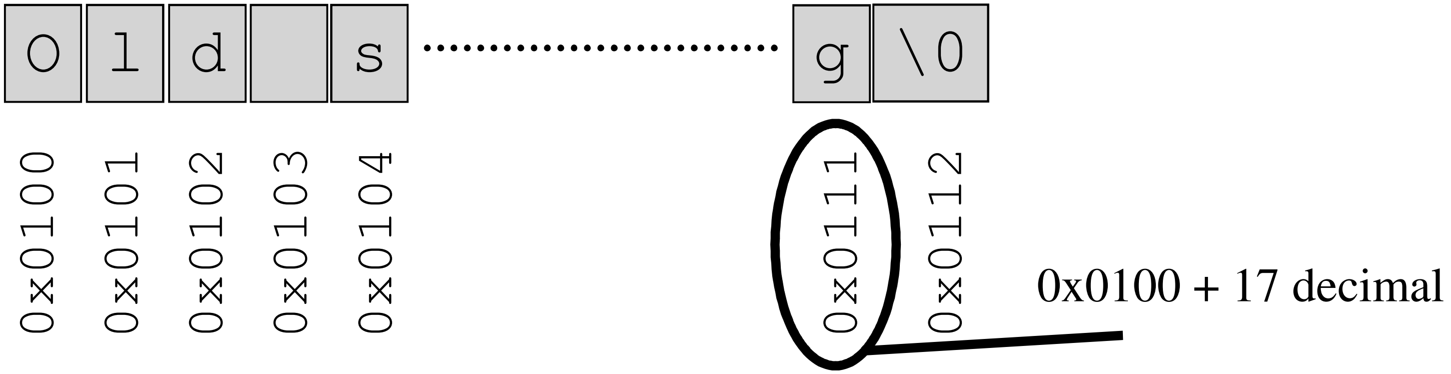
- – Zero terminated character arrays
- – Fixed size memory
- – Set of global functions to work with strings

- **Example character array:**

```
const int Length = 128;
char myWord[Length]; // Not initialized!
// avoid old style C strings
// if you have to use them, at least make them const
const char sentenceA[] = "Old ";
const char sentenceB[] = {"style "};
const char sentenceC[]{" C-string"};
std::cout << sentenceA << sentenceB << sentenceC <<
std::endl;
```

Pointers to old style C-strings

- **In general same as other arrays of integral type**
  - BUT truncation with trailing 0



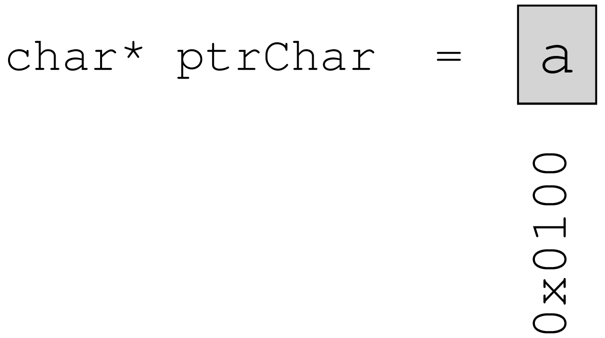
0x0100 + 17 decimal

```
char sentence[]{"Old style C-string"};
char* ptrSentence;
ptrSentence = &sentence[0];
cout << *ptrSentence << " to " << *(ptrSentence+17) << endl;
cout << std::hex << (size_t) ptrSentence << " to " <<
 (size_t) ptrSentence+18 << std::dec << endl;
```

Re-Interpretation Casts

- `reinterpret_cast`
  - Bit pattern of one variable is interpreted as another type: used with pointers; data bits do not change!
  - Dangerous: machine-dependent

```
char a = 'a'; char* ptrChar = &a; int* ptrInt;
ptrInt = reinterpret_cast<int*>(ptrChar);
```



Scope of Names

- **Local or block scope**
  - A name declared inside a block is accessible from the point of declaration to the end of the block.
- **Global (file) scope**
  - A name declared outside any function can be accessed inside the file from the point of declaration.
- **Class scope**
  - A name of a class element is local to the class, i.e., can only be accessed inside the class or must be used together with `.` `->` or `::`
- **Function scope**
  - Only for labels; accessible everywhere in the function
- **Prototype scope**
  - Names are only accessible in the prototype declaration

Example

- **Global scope**
  - PayRate
  - CalculateWage
- **Local scope (inside main)**
  - hours
  - pay
- **Local scope (inside CalculateWage)**
  - workHours
  - res

```
float PayRate = 1.5;
float CalculateWage(float);

int main( void )
{
 float hours;
 float pay = CalculateWage(hours);
 return 0;
}

float CalculateWage( float workHours )
{
 float res = workHours * PayRate;
 return res;
}
```

Scope, Storage class and Linkage

- Three different concepts
- Definitions and keywords are intertwined
  - Scope: Accessibility of a name
  - Storage class: Existence of the variable
  - Linkage: Known in current unit only or also in other translation units

Storage Class Modifiers

- **static**
  - Static declares a variable/function to have static duration. It is allocated at program (thread) initialization and remains accessible until the program (thread) exits.
  - A static variable inside a function is initialized once and remains unchanged between function calls.
  - Static data members of classes exist once per class and must be initialized within the same file scope.
  - static and extern are related but different

```
void myFunction()
{
 static int cnt = 0;
 cnt++;
 std::cout << “Call no.: “ << cnt << std::endl;
}
```

Storage Class Modifiers

- **extern**
  - extern declares that a global variable/function of static duration exists.
  - A extern variable may be initialized in the same file or in another file within the project.
  - Declarations in a different language need to be included with, e.g., extern “C” { … }

```
int cnt = 0;
void myFunction() {
 extern int cnt;
 cnt++;
 std::cout << “Call no.: “ << cnt << std::endl;
}
```

Differences between Extern and Static Linkage

- static names have internal linkage (Exception: static members of a class). Name is not visible outside the current translation unit.
- extern names have external linkage

Use	Static	Extern
Function declarations within a block	No	Yes
Names in a block	Yes	Yes
Names outside any block	Yes	Yes (default)
Functions	Yes	Yes (default)
Methods of a class	Yes	No
Attributes of a class	Yes	No

Storage Class Modifiers

- **const**
  - const declares that a variable, object is constant and will not change
  - Constant variables are especially important with pointers
  - Use const as much as possible

```
const int arrLength = 1000;
int myArray[arrLength];  // Allowed in C++!

int myFunction( const int myNum ) {
  res = 5 * myNum;
  myNum = 3; // Illegal!
}
```

Type Aliasing (`using` or `typedef`)

- `using` creates an alias to a data type.
- Works for fundamental, derived and composed data types, e.g., `int, enum, struct, union`
- Makes use of composed data types simpler

```
using Counter=int;
Counter i, j;
```

- using was introduced with C++11, prior there was `typedef`

```
typedef int Counter;
Counter i, j;
```

- `typedef` has awkward syntax and cannot be used with templates to define a family of types

Next 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