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++
- **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

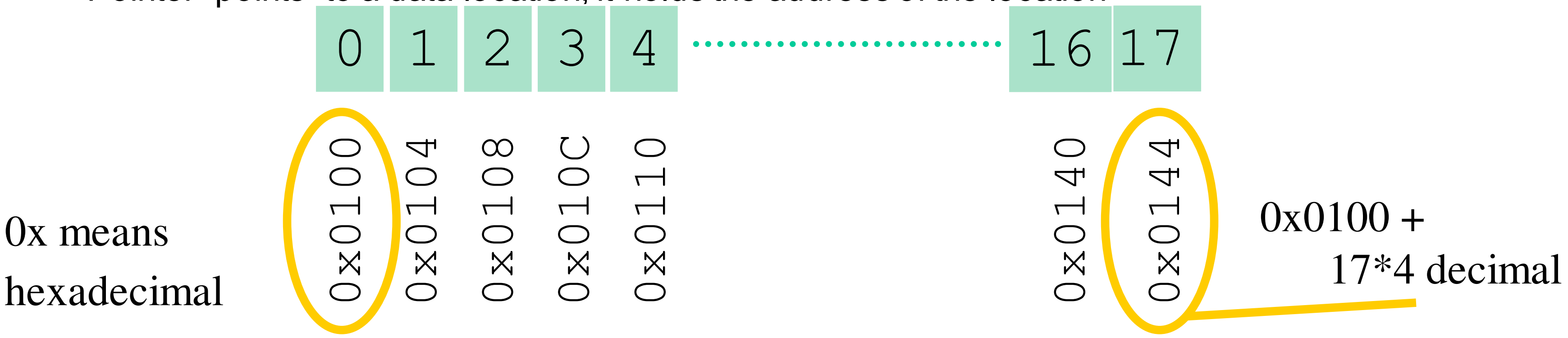
- **Purpose**
  - Accessing the address of a variable
  - Sending / passing a parameter by reference to a function
  - Parallel array / pointer allowing to pass the address of the first element of an array to a function
  - Passing a function as a parameter to a function
- **Pointers are iterators for arrays**
  - Pointers are fixed size and independent of data size
  - Pointers hold the (partial) address of memory
  - Operators for pointer arithmetic are : + ++ - -- = += -= ==

Is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory.

```
int num[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
              11, 12, 13, 14, 15, 16, 17 };
int *ptrNum; // Declaration of the pointer
ptrNum = &num[0]; // assigning the reference of the first elt
cout << *ptrNum << " to " << *(ptrNum+17) << endl;//0 to 17
cout << ptrNum << " to " << (ptrNum+17) << endl;//addresses
cout << "Array Length (Bytes): " << (size_t)(ptrNum+17) -
 (size_t) ptrNum + 1 * sizeof(int) << endl; // 72
```

Pointers

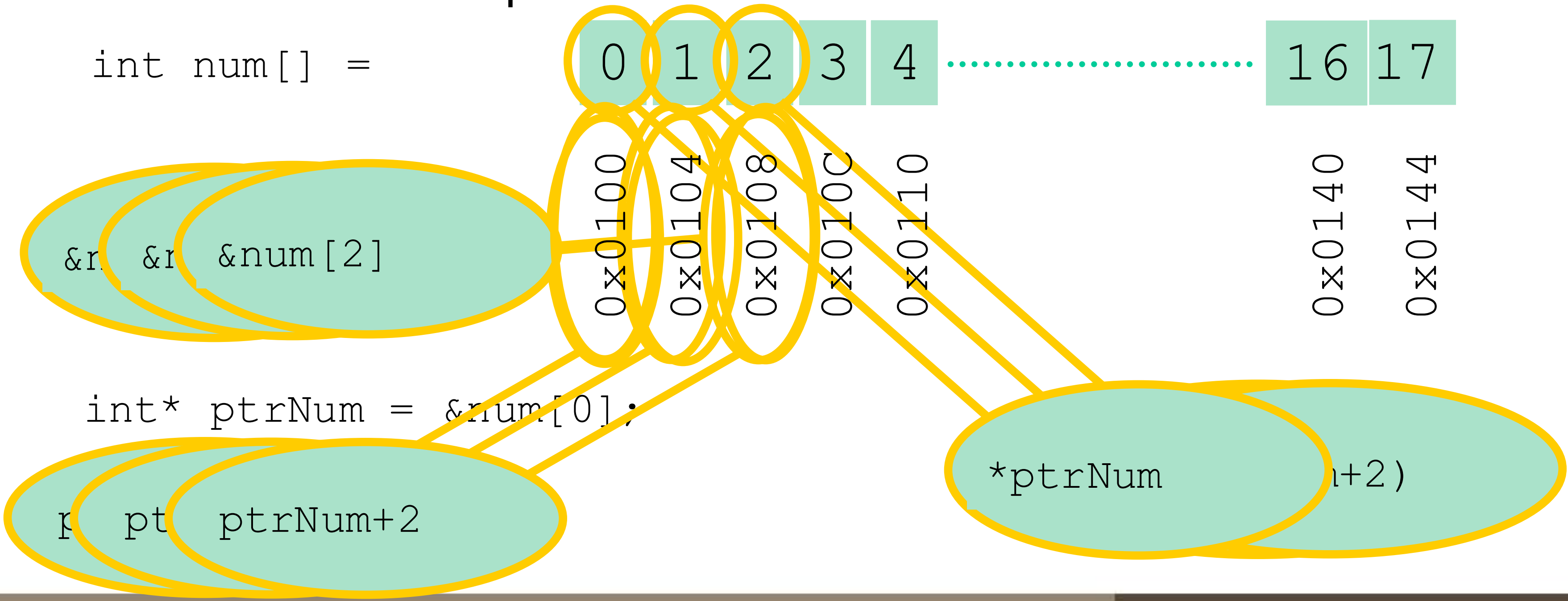
- **Purpose**
  - Accessing the address of a variable
  - Sending / passing a parameter by reference to a function
  - Parallel array / pointer allowing to pass the address of the first element of an array to a function
  - Passing a function as a parameter to a function
- **Data is located somewhere in memory**
  - Pointer "points" to a data location, it holds the address of the location



```
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

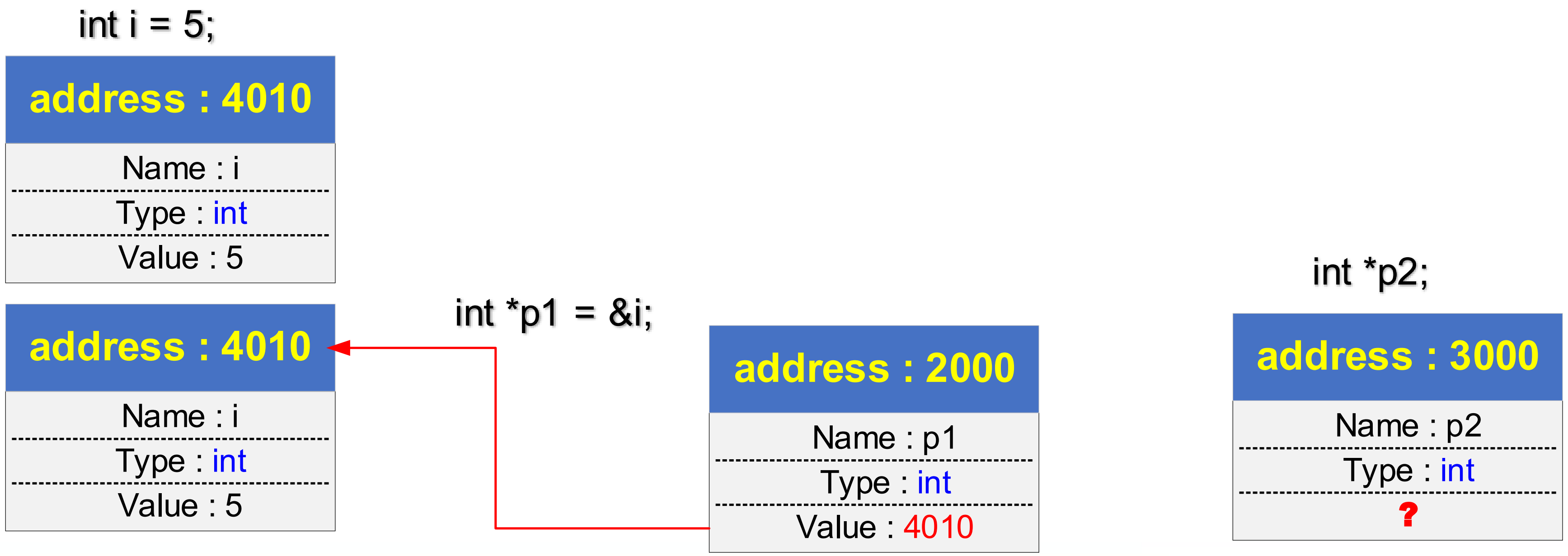
- **Operators**
  - Declaration of a pointer type * pointerToType
  - Address of operator &
  - Dereference operator * // " Content of "



Pointer , Address and Dereference

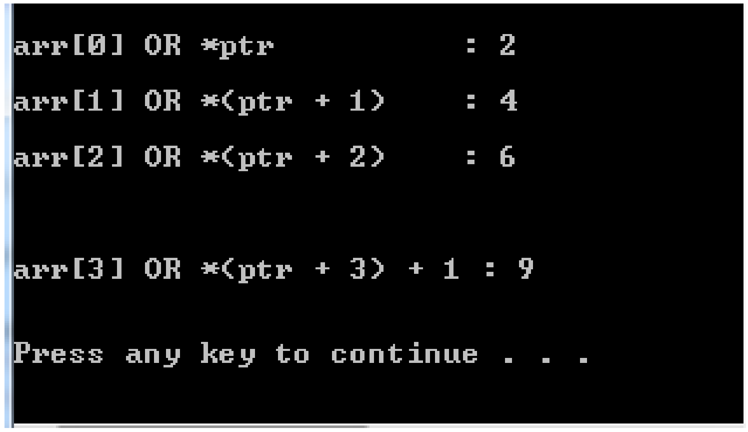
- **Example**

```
int main(){
    int i = 5;
    int *p1 = &i;
    int *p2;
    cout << "p1 =   "  << *p1 << endl; /* Display the contents of the object pointed to by p1 : 5 */
    cout << "p2 =  "  << *p2 << endl; /* Runtime error because the pointer p2 is not initialized */

    return 0;
}
```



Pointer , Address and Dereference

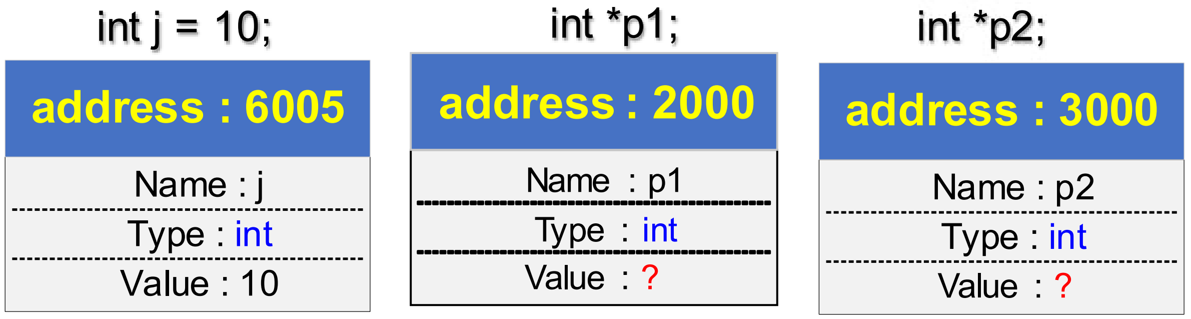
```
#include <iostream>
using namespace std;

int main() {
    int arr[5] = {2, 4, 6, 8, 10};

    int *ptr;

    ptr = &arr[0];

    cout << endl;
    cout << "arr[0] OR *ptr: " << *(ptr + 0) << endl << endl;
    cout << "arr[1] OR *(ptr + 1): " << *(ptr + 1) << endl << endl;
    cout << "arr[2] OR *(ptr + 2): " << *(ptr + 2) << endl << endl;

    cout << endl << endl;

    cout << "arr[3] OR *(ptr + 3) + 1 : " << *(ptr + 3) + 1 << endl;

    cout << endl << endl;
    system("PAUSE");
    return 0;
}
```

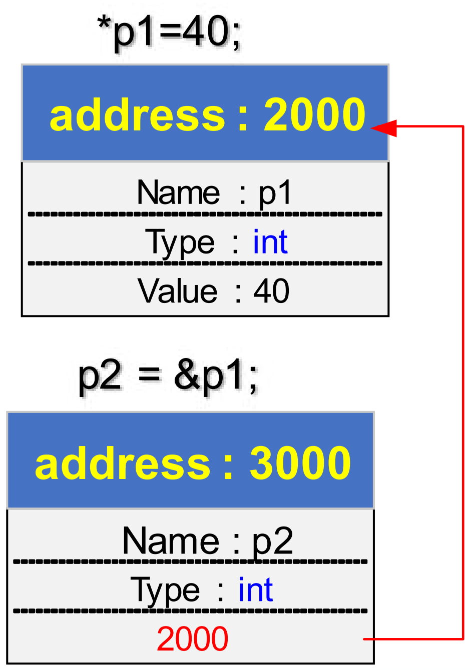


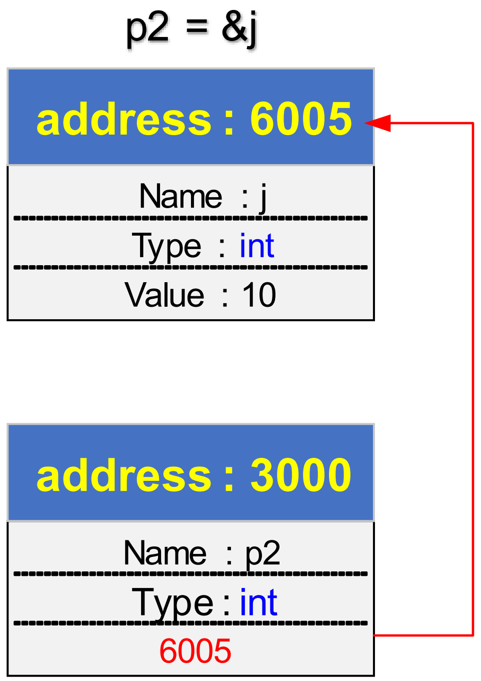
```
arr[0] OR *ptr           : 2

arr[1] OR *(ptr + 1)     : 4

arr[2] OR *(ptr + 2)     : 6


arr[3] OR *(ptr + 3) + 1 : 9


Press any key to continue . . .
```

Pointers

```
int j = 10;
int *p1, *p2;
*p1 = 40;
p2 = p1;
//*p2 = 30;
p2 = &j;
```







Pointers

- **Expressions containing pointers**

```
int size1(char *str[ ]) {
    int som = 0;
    char *p = *str;
    while (*(p++)) {
        som++;
    }
    return som;
}
```

```
int main(){
    char *str[ ] = {"Hello"};
    char *ptr = *str;

    int result = size1(&ptr);

    cout << "The size of the string is " << result << endl;

    cout << endl; cout << endl;
    system("PAUSE");
    return 0;
}
```

Output:

```
The size of the string is 5

Press any key to continue …
```

Pointers

- **The pointer parameters to a function**

```
void MultiService(int *a, int *b, int *c)
{
    int t = *a, k = *b;
    *a = t + k;
    *b = t - k;
    *c = t * k;
    cout << "The value of a = " <<  *a << " , b = " << *b << " , and c = " <<  *c << endl;
}

void main()
{
    int x = 5, y = 3, z;

    MultiService(&x, &y, &z);

    cout << endl; cout << endl;
    system("PAUSE");
}
```

Output:

```
The value of a = 8 , b = 2 , and c = 15
```

Pointers

- **The pointer to a function**

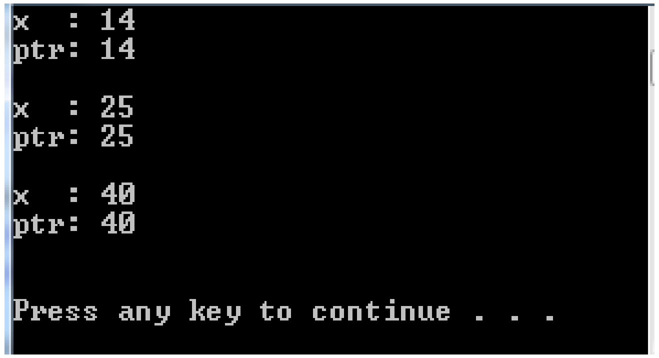
```
#include <iostream>
using namespace std;

int min(int a, int b)
{
    return a < b ? a : b;
}

int max(int a, int b)
{
    return a > b ? a : b;
}


int sum(int a, int b)
{
    return a + b;
}
```

```
void main()
{
    int a = 5, b = 4;

    int (*ptr1)(int, int);
    ptr1 = &min;
    int smaller = ptr1(a,b);

    int (*ptr2)(int, int);
    ptr2 = &max;
    int bigger = ptr2(a,b);

    int (*ptr3)(int, int);
    ptr3 = &sum;
    int sum = ptr3(a,b);

    cout << "Smaller = " << smaller << " , Bigger = " << bigger <<  " , Sum = " " << sum);

    cout << endl; cout << endl;
    system("PAUSE");
}
```

Pointers

- Aliasing

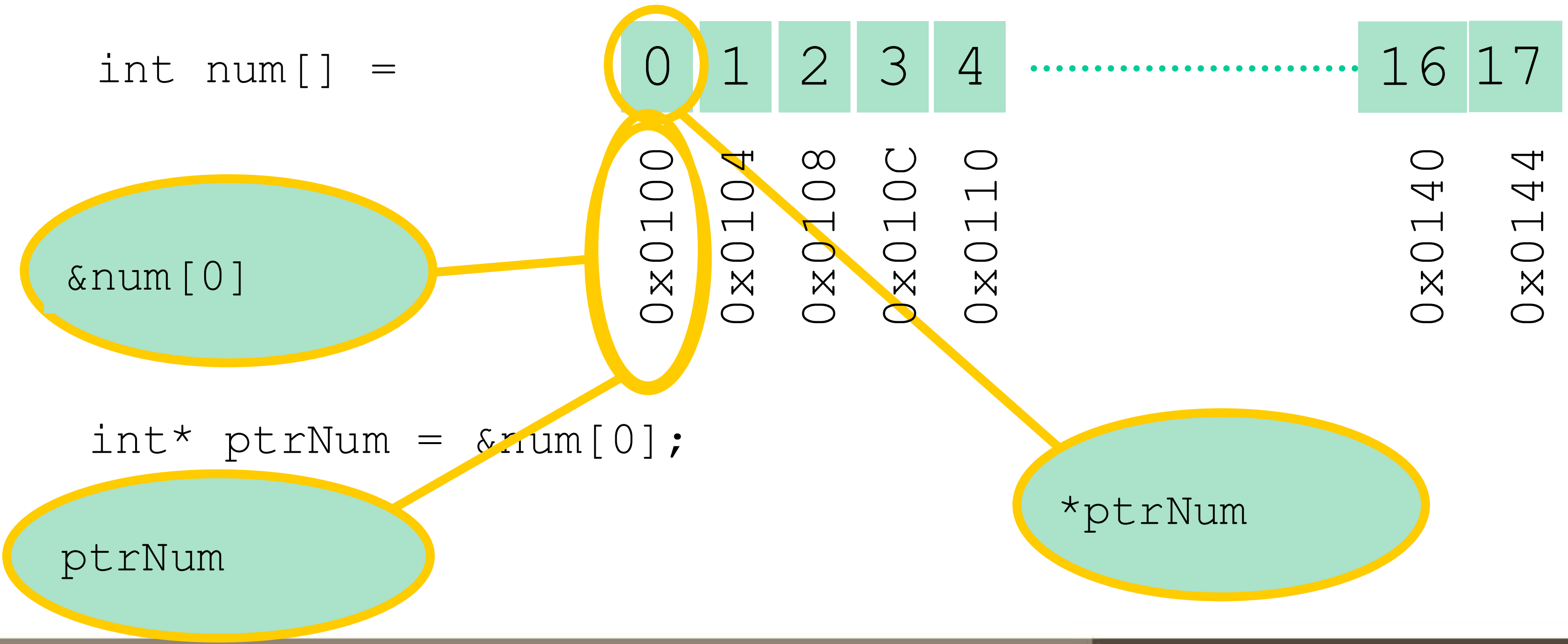
```
#include<iostream>
using namespace std;

int main() {
    int x = 14;
    int *ptr;

    ptr = &x;

    cout << "x  : " << x <<  endl;
    cout <<"ptr: " << *ptr << endl << endl;

    x = 25;
    cout << "x  : " << x <<  endl;
    cout << "ptr: " << *ptr << endl << endl;

    *ptr = 40;
    cout << "x  : " << x << endl;
    cout << "ptr: " << *ptr << endl << endl;

    cout << endl;
    system("PAUSE");
    return 0;
}
```



```
x   : 14
ptr: 14

x   : 25
ptr: 25

x   : 40
ptr: 40


Press any key to continue . . .
```

Address and Dereference (or "Content of")

- **Operators**
  - Declaration of a pointer `type * pointerToType`
  - Address of operator `&`
  - Dereference (or "Content of") operator `*`



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; // 3

// Access through pointer to elements
int* firstE = &numbers[0];
cout << firstE[3] << endl; // 3

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

Old-style C-strings

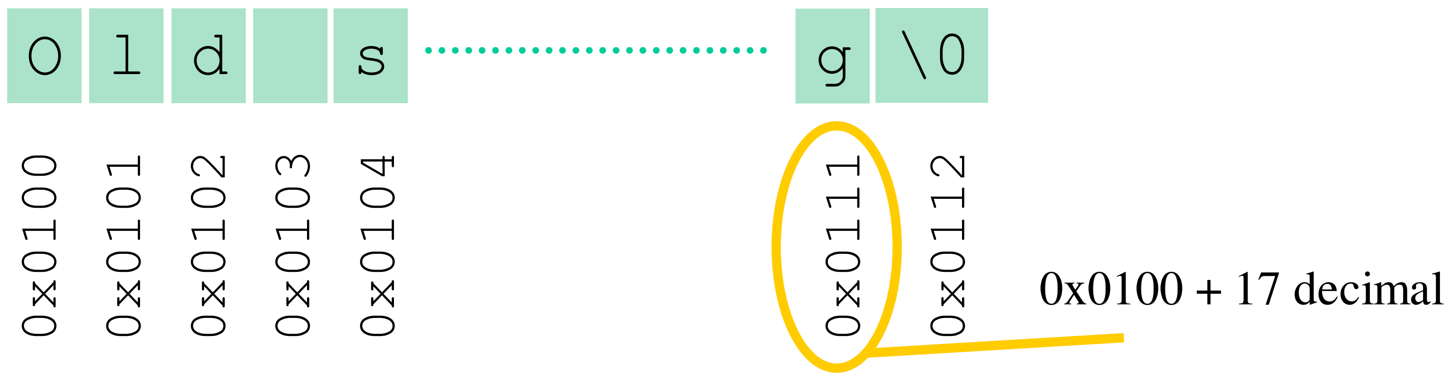
  – Zero terminated character arrays
  – Fixed size memory
  – Set of global functions to work with strings

- **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"};

cout << sentenceA << sentenceB << sentenceC << endl; // Old style C-string
```

Pointers to old style C-strings

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



```
char sentence[] = {"Old style C-string"};
char* ptrSentence;
ptrSentence = &sentence[0];
cout << *ptrSentence << " to " << *(ptrSentence+17) << endl; // o to g
cout << std::hex << (size_t) ptrSentence << " to " <<
  (size_t) ptrSentence+18 << dec << endl; // 1cf848 to 1cf85a
```

Pointers

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

int main() {

    char text[]  = "Hello";

    char *adr = text;

    cout << "Give a character : ";
    cin >> *adr;

    cout << endl;

    do{
          cout << "*adr = " << *adr << endl;
     } while ( *(adr++) );

    cout << endl;
    system("PAUSE");
    return 0;
}
```

InitAssPointerVar.cpp

```
int size(char str[ ])
{
     int i = 0;
     char *ptr = str, *ptr1 = str;          /* str[0] ⇔ *(ptr + 0) */
     while (*(ptr++)) {                     /* str[1] ⇔ *(ptr + 1) ⇔ *(ptr++) */
        i++;
     }
      return i;                            /* ⇔ return ptr - ptr1; */
}
```

*(ptr + 1) is not the same as *(ptr) + 1.

*(ptr + 1) is the same as arr[1].

*(ptr) + 1 is the same as arr[0] + 1.

Pointers

- **The array of pointers**
  - **Pointers can be stored in an array in the same way as for integers, characters, and so on.**

arr[0][0] arr[0][5]

10	2	3	0	7
1	15	30	2	18
21	40	11		

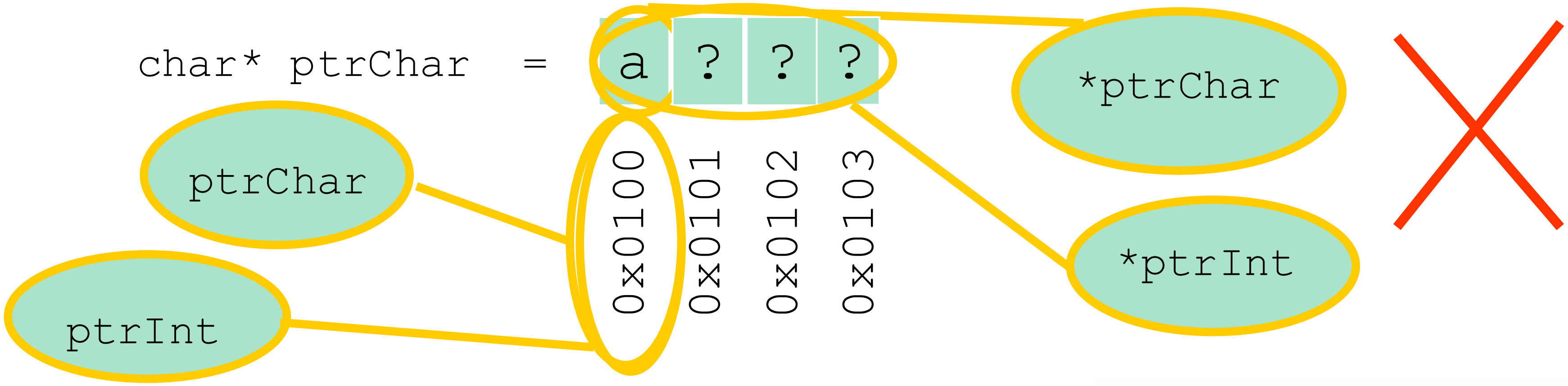
```
int arr[ ][5] = {10, 2, 3, 0, 7, 1, 15, 30, 2, 18, 21, 40, 11};
int *arr1 = arr[1], *arr2 = arr[2];
int *arrp[3]; /* ⇒ Array arrp of 3 pointers. Whereas int (*arrp)[3]: a pointer to an array of 3 integers */

arrp[1] = arr1;
arrp[2] = arr2;

int *pp = *arr;              /* *pp is the address of the array arr[0]. */
```

*(arrp + 3) + 2 : a pointer to the 3rd integer of the 2nd array ≡ arrp[3][2]

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

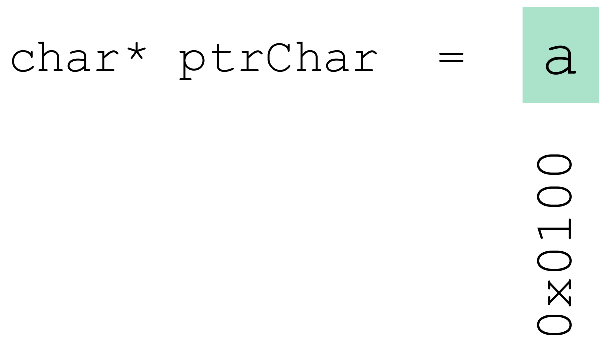
```
char a = 'a'; char* ptrChar = &a; int* ptrInt;
ptrInt = reinterpret_cast<int*>(ptrChar);
```



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;
}
```

Storage Class Modifiers

Ex1:

```
int var;
int main(void)
{
    var = 10;
    return 0;
}
```

Analysis : This program is compiled successfully. Here var is defined (and declared implicit) globally.

Ex2:

```
extern int var;
int main(void)
{
      return 0;
}
```

Analysis : This program is compiled successfully. Here var is declared only. Notice var is never used so no problems.

Ex3:

```
extern int var;
int main(void)
{
    var = 10;
    return 0;
}
```

Analysis : This program throws error in compilation. Because var is declared but not defined anywhere. Essentially, the var isn't allocated any memory. And the program is trying to change the value to 10 of a variable that doesn't exist at all

Storage Class Modifiers

Ex4:

```
#include "somefile.h"
extern int var;
int main(void)
{
    var = 10;
    return 0;
}
```

Analysis : Supposing that "somefile.h" has the definition of var. This program will be compiled successfully.

Ex5:

```
extern int var = 0;
int main(void)
{
        var = 10;
        return 0;
}
```

Analysis : Guess this program will work? Well, here comes another surprise from C standards.
They say that...if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated, i/e/, that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work.

In short, we can say:

1. Declaration can be done any number of times but definition only once.
2. "**extern**" keyword is used to extend the visibility of variables / functions.
3. Since functions are visible throughout the program by default. The use of **"extern"** is not needed in function declaration / definition. Its use is redundant.
4. When **"extern"** is used with a variable, it's only declared not defined.
5. As an expression, when an **"extern"** variable is declared with initialization, it is taken as definition of the variable as well.

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

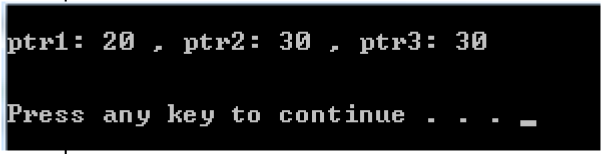
```
const int arrLength = 1000;
int myArray[arrLength];  // Allowed in C++!

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

Make Use of const modifier

```
const char * string1 = "Annie";        // string1 is declared on a constant character
char * const string2 = "Remi";         // string2 is declared on the constant pointer to a character
const char * const string3 = "Julie"; // a constant pointer to a constant character

string1[3] = 'e';          // Error : the value is a constant
string1 = string2;         // OK. the pointer is not constant so we can exchange the value

string2[2] = 'n';          // OK. The value is not a constant
string2 = string1;         // Error : string2 is a constant pointer and string1 is a constant character

string3[1] = 'o';          // Error : the value is a constant
string3 = string1;         // Error : string3 is a constant pointer and string1 is a constant character.

int i = 5, j = 2;          // OK
int * const p1;            // Error : p1 must be initialized because p1 is constant pointer to integer
const int * p2 = &i;       // OK

p2 = &j;      // OK.
*p2 = 10;     // Error : we cannot modify the constant of p2 because p2 is a constant
p1 = &i;      // Error : p1 is a constant pointer but &i is a reference of the constant i.
*p1 = 10;     // OK.
```

Constant pointers and pointers to constants

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

int main() {
      int x = 20;
      int const y = 30;

      // a constant pointer of integer type */
      int * const ptr1 = &x;

      // a pointer of constant integer type */
      int const* ptr2 = &y;

      // a constant pointer of constant integer type */
      int const* const ptr3 = &y;

      cout << endl;
      cout << "ptr1: " << *ptr1 << " , ptr2: " << *ptr2 << " , ptr3: " << *ptr3 << endl;

      cout << endl << endl;
      system("PAUSE");
      return 0;
}
```



```
ptr1: 20 , ptr2: 30 , ptr3: 30

Press any key to continue . . . _
```

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;
```

- Creating a type alias with **`using`** has exactly the same <u>same effect </u>as creating a type alias with **`typedef`**. It is simply an alternative syntax for accomplishing the same thing.
- **`using`** can be used with "Templates".
- **`typedef`** has awkward syntax and with templates to define a family of types. cannot be used

Example : typedef

```
struct T_Point
{
    float x-axis,  y-axis;
}

typedef struct T_Point Point;
Point p1, p2, p3;

typedef struct
{
    char lastname[21],
    firstname[21],
    pCode[13];
} Student;
```

```
struct DATE
{
    int day, month, year;
}

typedef struct DATE Date;
OR

struct Date
{
    int day, month, year;
}
typedef struct Date Date;
```

```
typedef unsigned short boolean;
boolean finds, exists;
```

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