Cairo University, Faculty of Computers and AI

CS213 – 2022 / 2023

Object Oriented Programming

Lecture 1: C++ Pointers

By

Dr. Mohammad El-Ramly

http://www.acadox.com/class/64401 PY7OGJ

Cairo University, Faculty of Computers and AI

CS112 – 2021 / 2022 2nd Term

Structured Programming

Lecture 11:
C++11 Pointers

By

Dr. Mohammad El-Ramly

Lecture Objective / Content

1. Pointers
2. Relation between Arrays and Pointers
3. Dynamic Memory Allocation
4. Smart Pointers
5. Reference Variables

Good Coding Style

- Any fool can write code the computers can understand. Only good programmers write code that humans can understand.

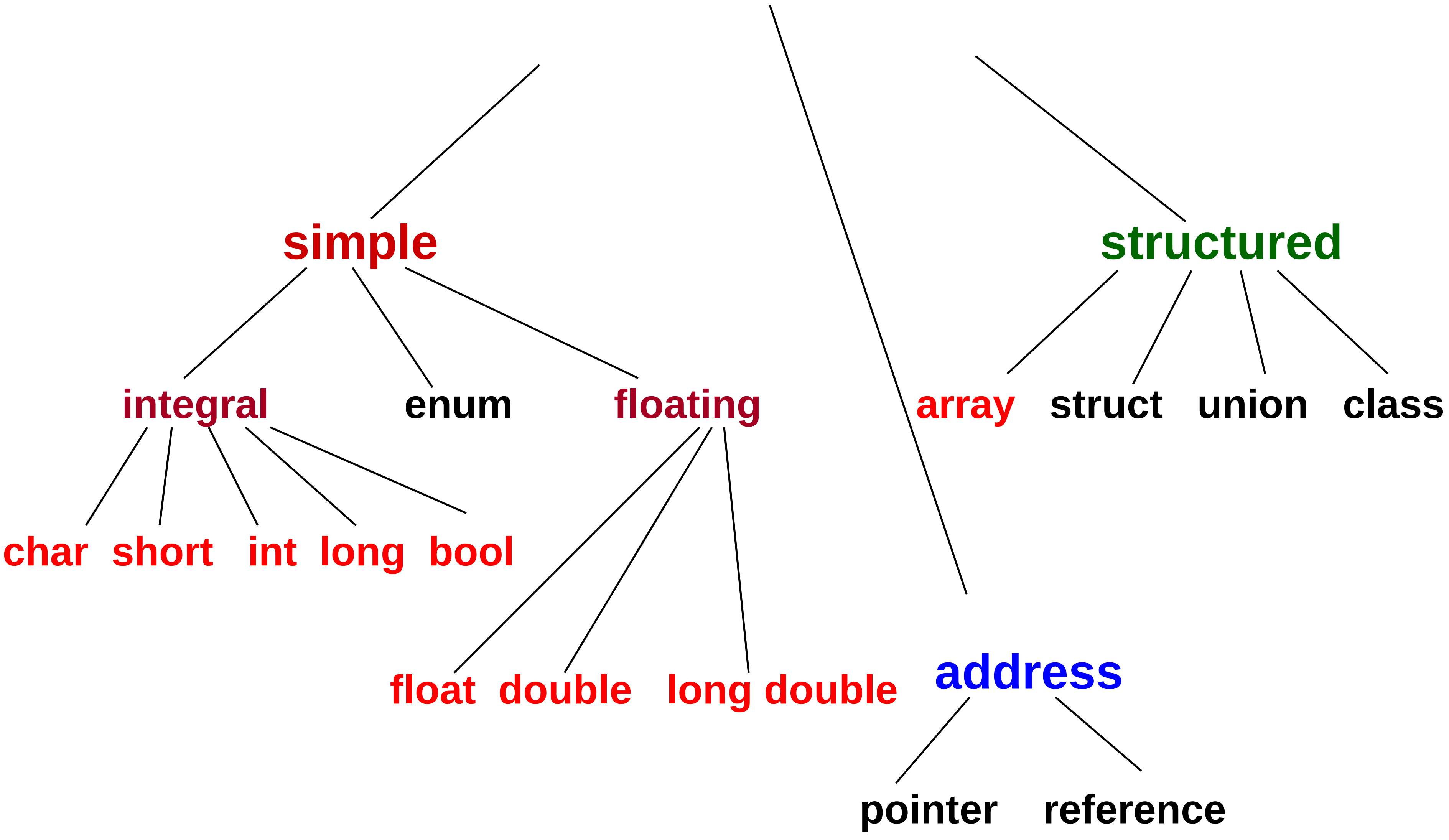
Martin Fowler

1. Choose a style guide to follow
2. Choose a naming convention and stick to it.

Good Coding Style

- Variable and function names are
  - All small separated by _ like **`total_size`**
  - Mixed starting by capital, except first like **`isValid`** and **`totalSize`**
- Constants are all capital like PI, SIZE
- Always leave space around operators except between [ ] like **`x = 5 + y;`**
- Use meaningful identifiers
- Code must be indented
- DO NOT put 2 statements on the same line
- Use { } with all conditional statements, even if you have one statement

C++ Data Types

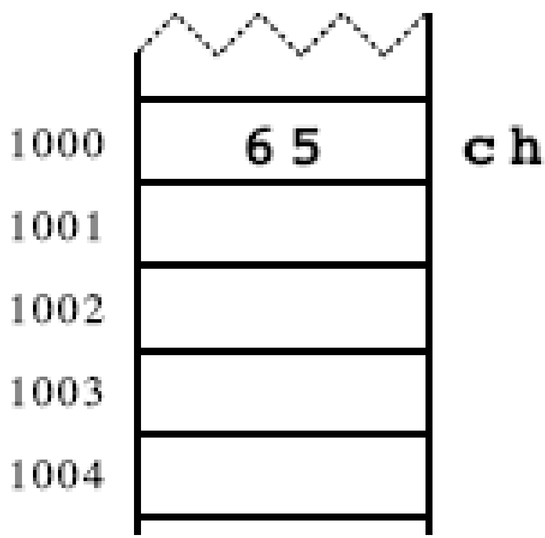

simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

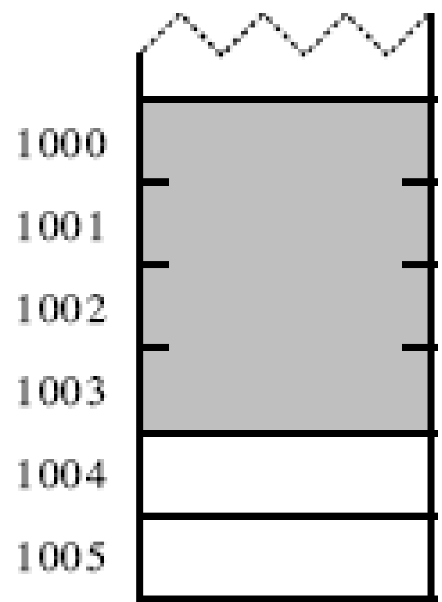
More Data Types

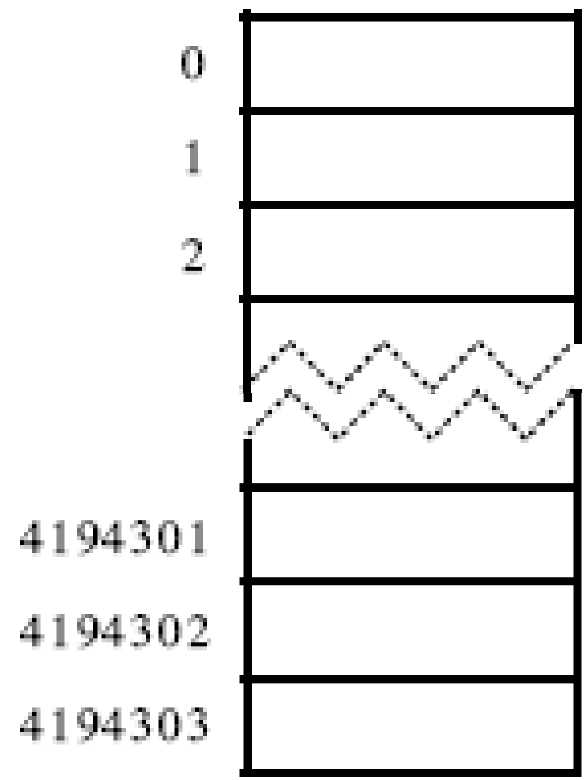
- Enumerations are types with restricted set of possible values.
- Pointers are the internal addresses of a value in the memory
- Arrays are ordered collections of data of the same type
- Records / Structures are collections of data, each consists of a items of different types that represent a coherent whole

Memory Representation of Data

- Memory is divided to bytes and words
- Programs are compiled with relative addresses and then when loaded to memory, addresses are adjusted.
- **`sizeof(int)`**
- **`sizeof x`**







1. Pointers

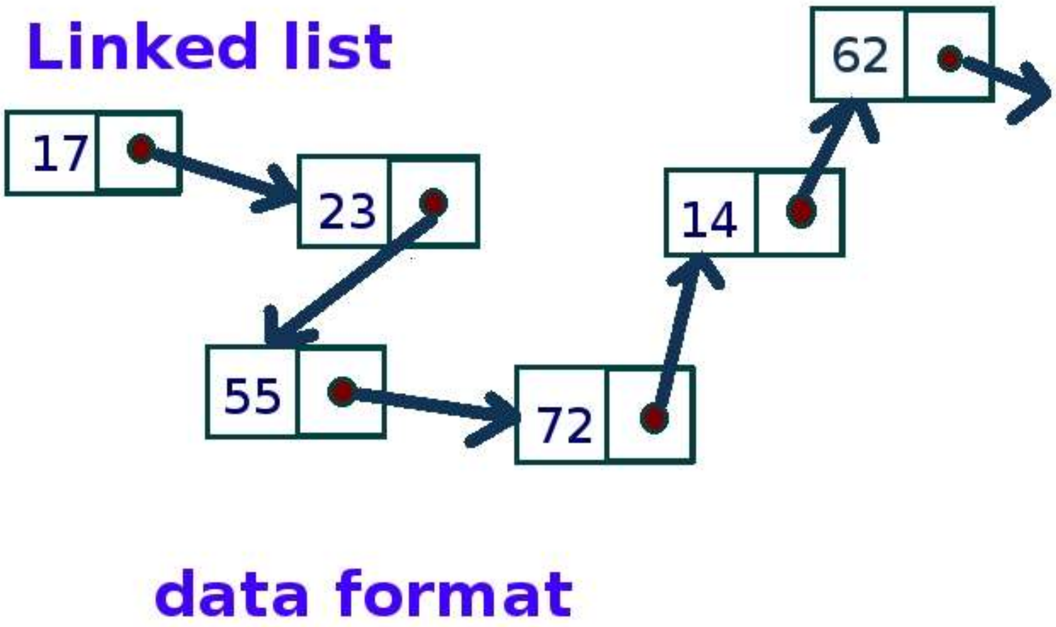
- The programmer should have as much access to the machine hardware as possible.
- Memory addresses available to the programmer.
- A variable can hold the memory address of a data value. Such a variable is called a pointer.
- Using pointers is like dancing with the elephants
- Java chose to leave this feature out.

Bjarne Stroustrup

- C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off

Why Pointers?

- They allow you to reserve new memory during runtime. (Dynamic allocation)
- Pointers allow you to refer to a large data structure in a compact way
- They are used to link individual data items in data structures, e.g., linked lists or trees.

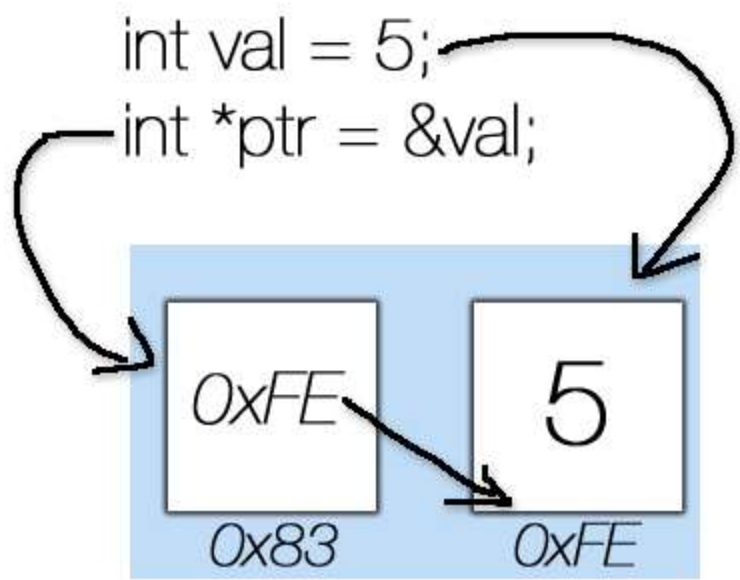


Addresses as Data Value

- An assignment has the format ***`lvalue = rvalue`***
- An ***`lvalue`*** is:
  - Stored in memory
  - Its location in memory does not change
  - Requires a certain amount of memory depending on the data it stores
  - Its address is a pointer value that can stored in another location in memory.

Declaring Pointers

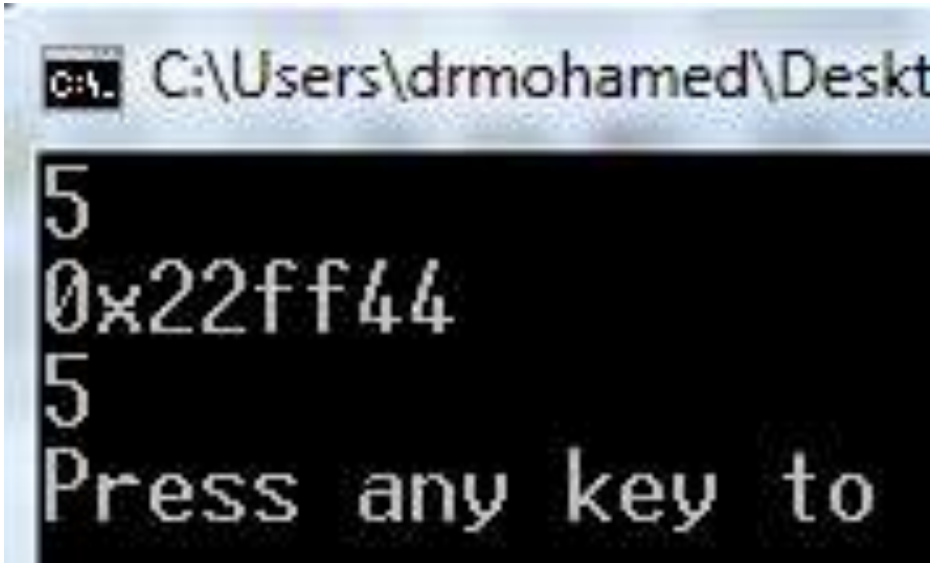
- `int val = 5;`
- `int *ptr = &val;`



- `int *p1;          // Create 1 pntr`
- `int *p1, *p2;     // Create 2 pntrs`
- `int *p1, p2;      // Creates ??`

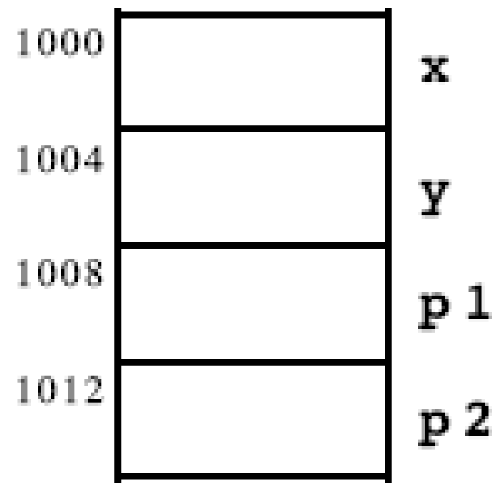
Pointer Operators: `*`, `&`, `=`

- `int val = 5;`
- `int* ptr = &val;`
- `cout << val;     // prints 5`
- `cout << ptr;     // prints address`
- `cout << *ptr;`

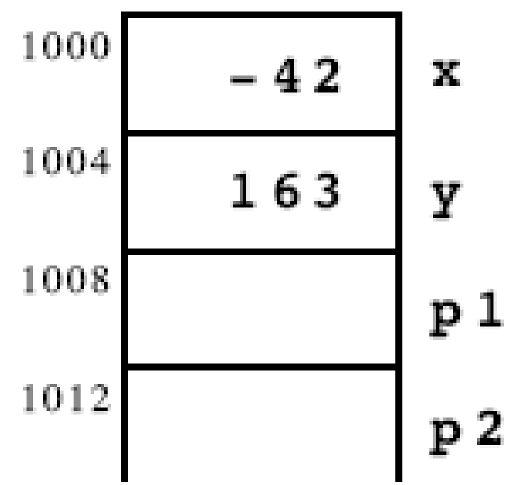


Pointer Operators: `*`, `&`, `=`

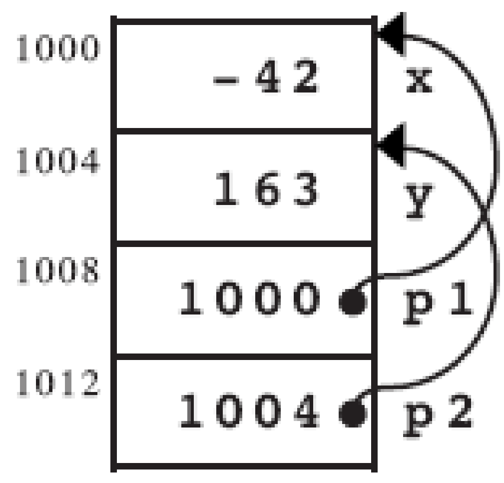
- `int x, y;`
- `int *p1, *p2;`



- `x = -42;`
- `y = 163;`

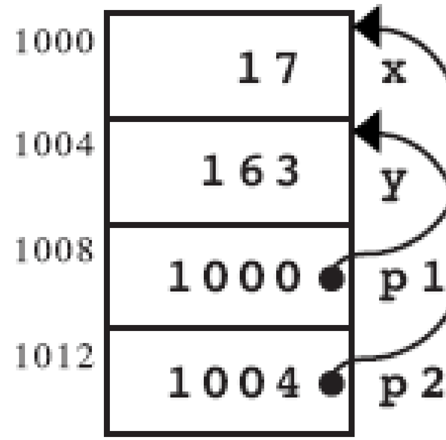


- `p1 = &x;`
- `p2 = &y;`

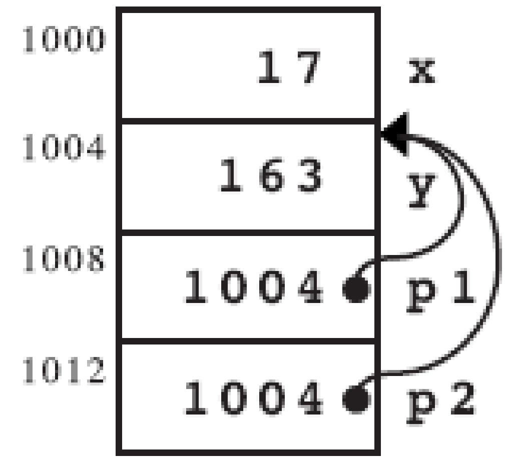


Pointer Operators: `*`, `&`, `=`

- `*p1 = 17;`

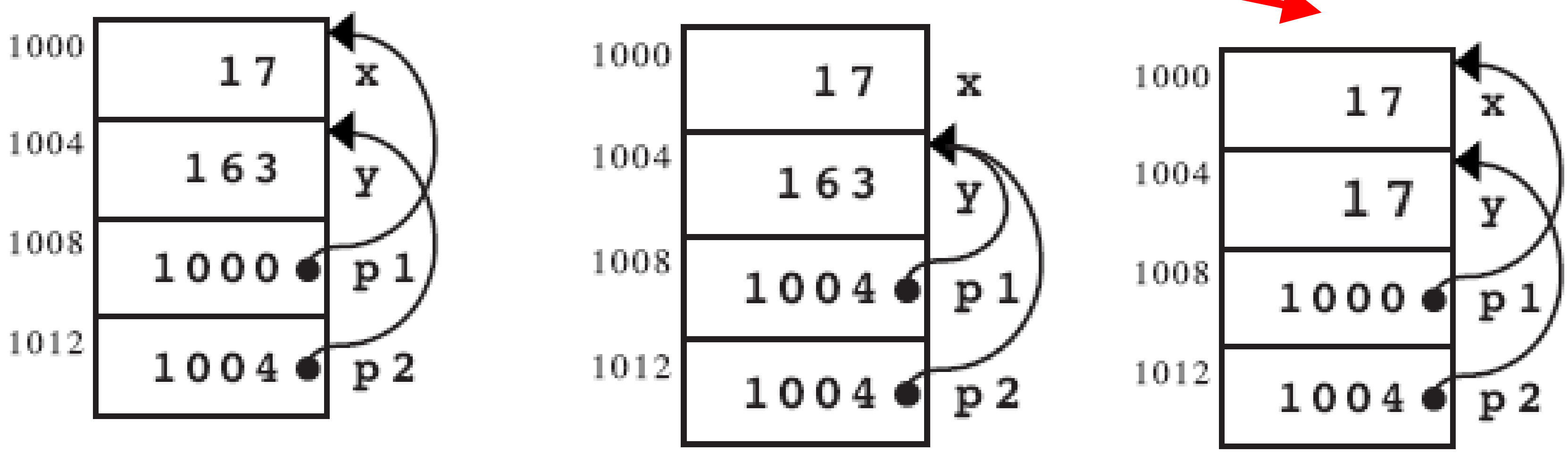


- `p1 = p2`



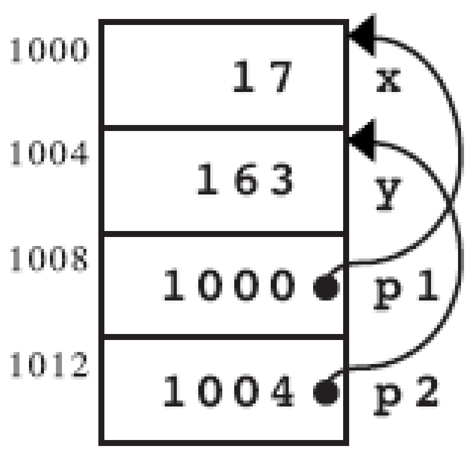
Pointer Operators: `*`, `&`, `=`

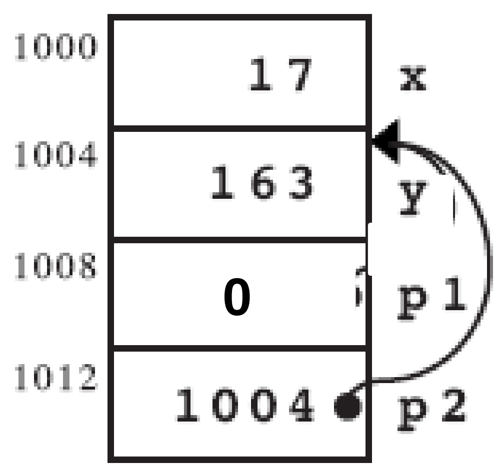
- `p1 = p2;`
- `*p2 = *p1;`



Pointer Operators: `*, &, =`

- `ptr = NULL;`
- `cout << ptr;  // prints 0`
- `cour << *ptr; // crash`





2. Arrays

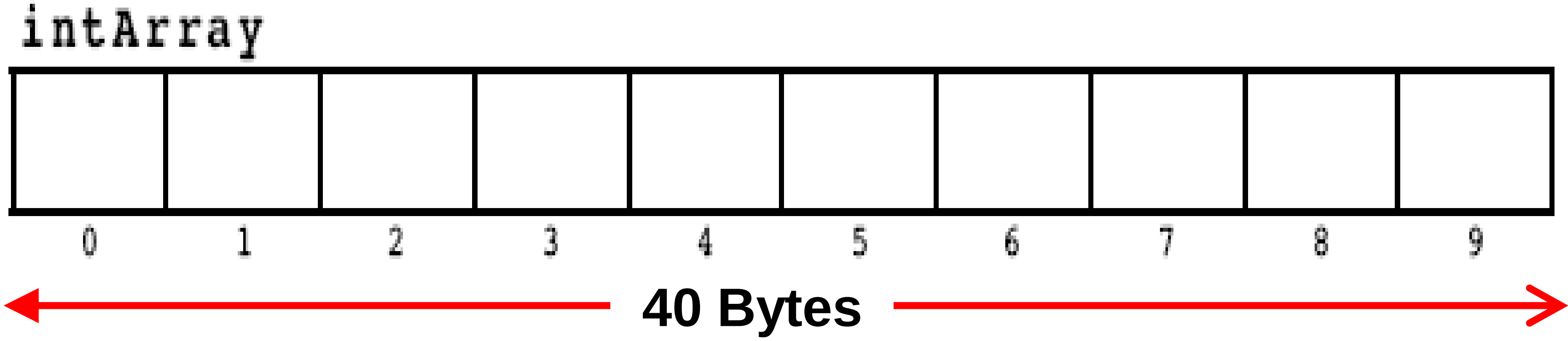
- An array is an indexable structure that consists of items of the same type.
- Items have positions.
- Array attributes:
  - Element type
  - Array size

- Array Declaration
  - *type name***[size];**

Array Declaration

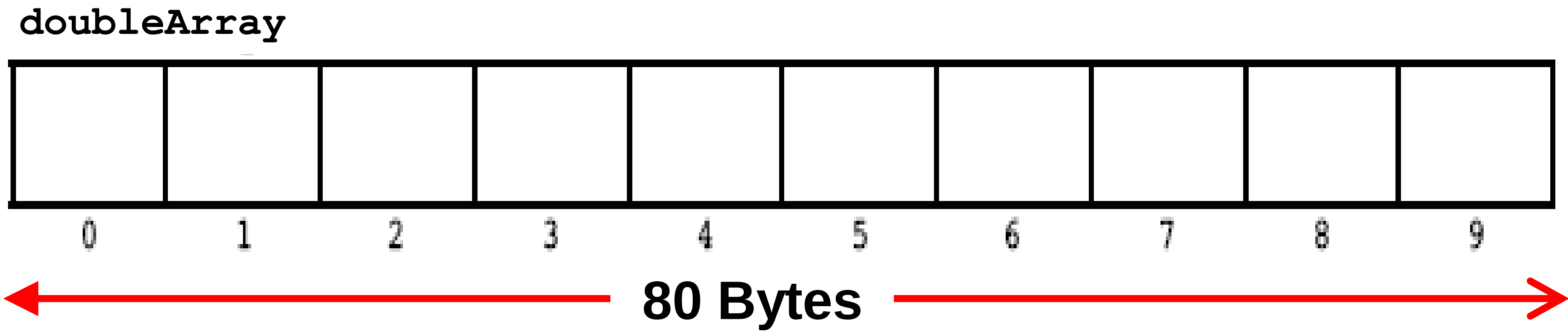
- `int intArray[10];`

- `const int N_ELEMENTS = 10;`
- `int intArray[N_ELEMENTS];`



Array Declaration

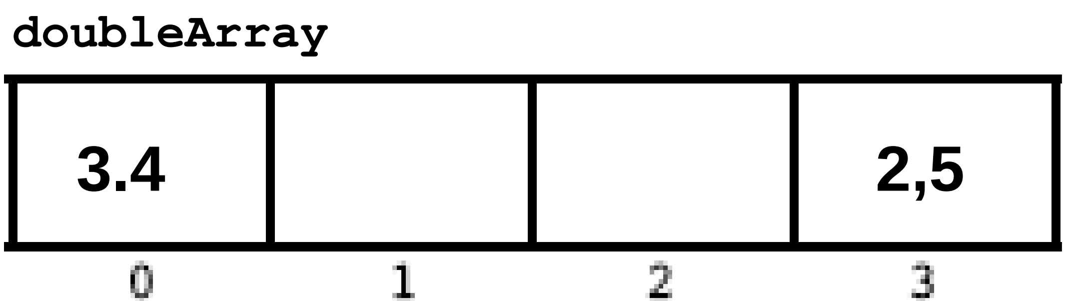
- `const int N_ELEMENTS = 10;`
- `double doubleArray[N_ELEMENTS];`



Referencing Array Cells

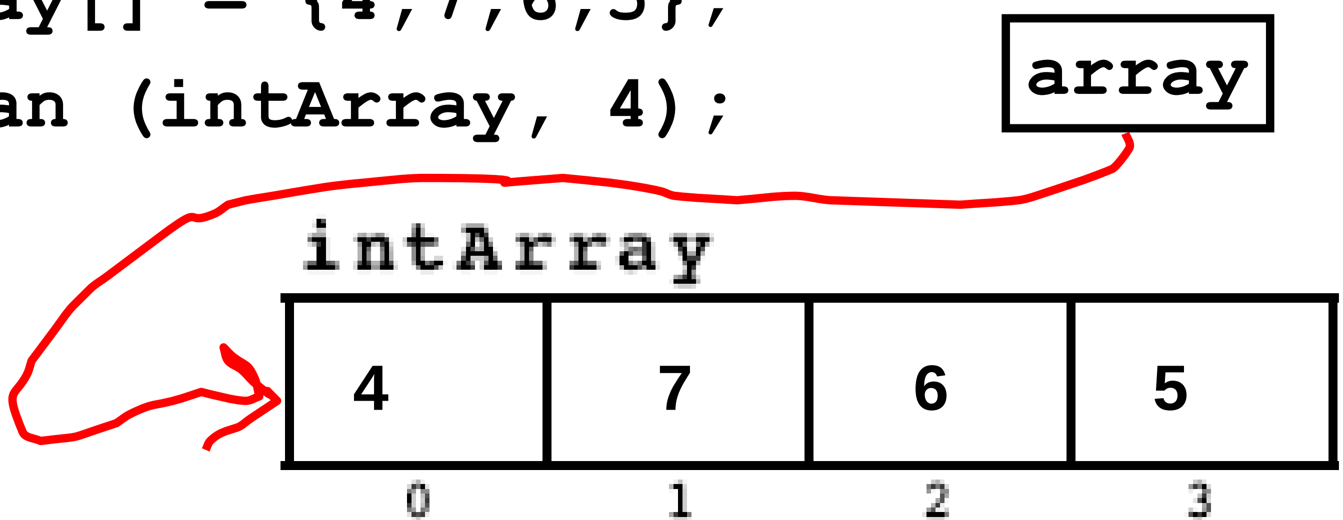
- `const int N_ELEMENTS = 4;`
- `double doubleArray[N_ELEMENTS];`
- `doubleArray[0] = 3.4;`
- `doubleArray[3] = 2.5;`
- `doubleArray[4] = 1.1;`

`// What will happen?`



Passing Arrays to Functions

```
double Mean(int array[], int n) {
     double total = 0;
     for (int i = 0; i < n; i++)
          total += array[i];
}
....................
int intArray[] = {4,7,6,5};
cout << Mean (intArray, 4);
```



Getting Array Size

- To write arrays whose length is defined by the data supplied, we need to discover the length automatically.

- **`string bigCities[] =`**

  `{"Jeddah", "Mekkah", "Almadinah"};`

- **`int nBigCities =`**

  `sizeof bigCities / sizeof bigCities[0];`

Pointers and Arrays

- The name of an array represents the address of the first entry.
  - Same as a pointer.

- A pointer can be set to the address of an array.

```
int values[20];
int *pValues;
...
pValues = values;
```

Pointers and Arrays

- Pointers and array names can be used interchangeably.
- When a pointer holds the base address of an array, we can put an index after it to refer to any element of the array.

```
pValues[10] = 201;
```

- Same effect as

```
values[10] = 201;
```

Pointers vs Arrays

- Array is an address constant that points to the same address always.
- **`int main () {`**
- **`   int arr1[4] = {1,2,3,4};`**
- **`   int arr2[4] = {1,2,3,4};`**
- **`   arr1[0] = 11;              // OK`**
- **`   //arr = {6,7,8,9}; // WRONG`**
- **`   //arr1 = arr2;       // WRONG`**
- **`   //arr points to same place`**
- **`}`**

Pointers vs Arrays

- Pointers are address variables that point to variable values
- `int main () {`
- `    int* parr1 = new int[4] {1,2,3,4};`
- `    int* parr2 = new int[4] {1,2,3,4};`
- `    parr1[0] = 11;                     // OK`
- `    parr1 = new int[4] {6,7,8,9};// OK`
- `    parr1 = arr2;                      // OK`
- `}`

Adding `const` to pointer

- `const int* ptr = new int(4);`
- `ptr = new int(44);`
- `// *ptr = 10; // WRONG const value`

- `int* const ptr2 = new int(4);`
- `*ptr2 = 44;`
- `// ptr2 = ptr // WRONG const ptr`

Adding `const` to pointer

- `// const pointer to const value`
- `const int* const x = new int(4);`
- `//x  = new int(4);    // WRONG`
- `//*x = 333;           // WRONG`

Using a pointer with an index

```
#include <iostream>

int main ( )
{
    int some_numbers[] = {101, 102, 103, 104, 105};
    int length = sizeof(some_numbers) /sizeof(some_numbers[0]);
    int i;
    int* pNumber = some_numbers;

    for (i = 0; i < length; i++)
    {
        cout << "Entry " << i << " contains " << pNumber[i];
    }
    return 0;
}
```

Same as some_numbers[i]

Using a pointer with an index

```
Entry 0 contains 101
Entry 1 contains 102
Entry 2 contains 103
Entry 3 contains 104
Entry 4 contains 105
```

Pointer Arithmetic

- We can also increment and decrement a pointer.

```
int values[20];
int* pValues;
...
pValues = values;
```

pValues points to values[0]

```
pValues += 1;
```

pValues points to values[1]

- The compiler knows the size of whatever the pointer points to and increments the address in the pointer appropriately.

Pointer Arithmetic

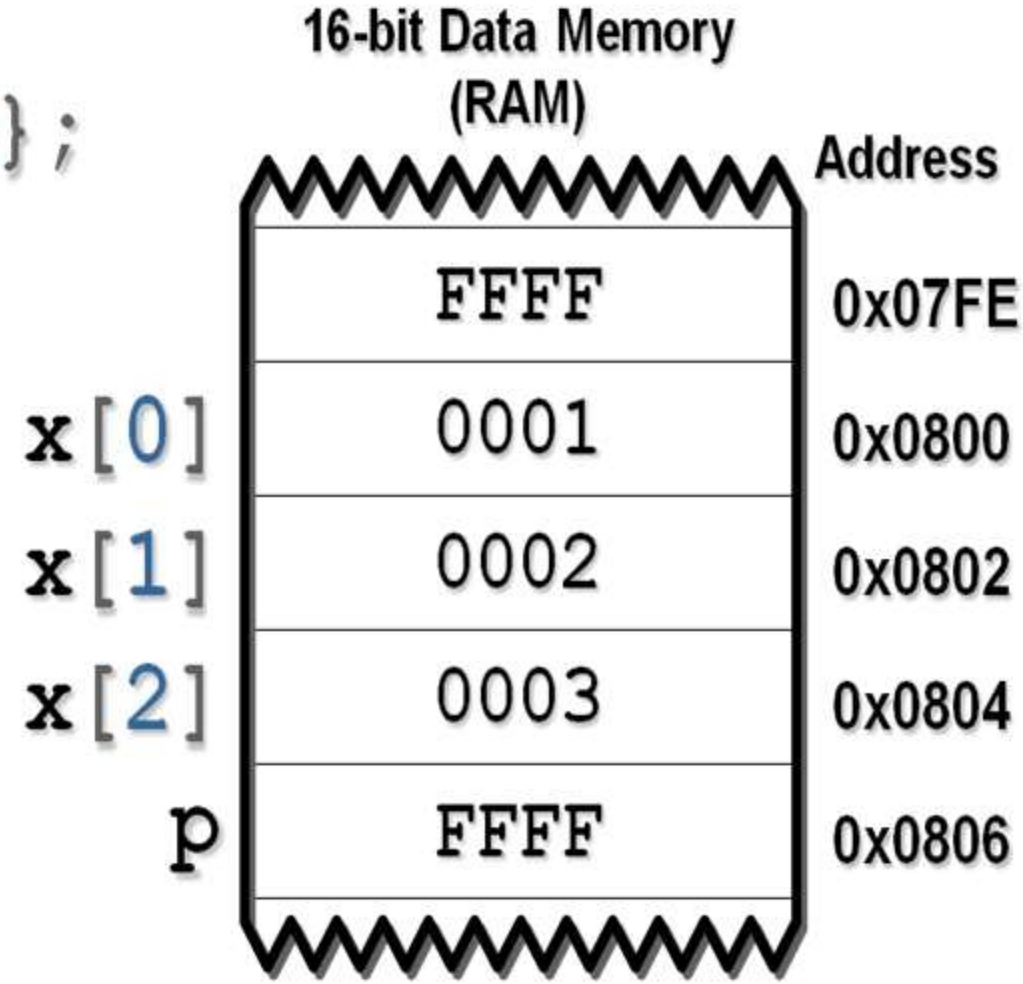
- Incrementing and decrementing a pointer only makes sense when the pointer points to an array.

- Increment says move forward that many entries.
- Decrement says move back that many entries.

- There is no check that the result is a valid reference to the array!

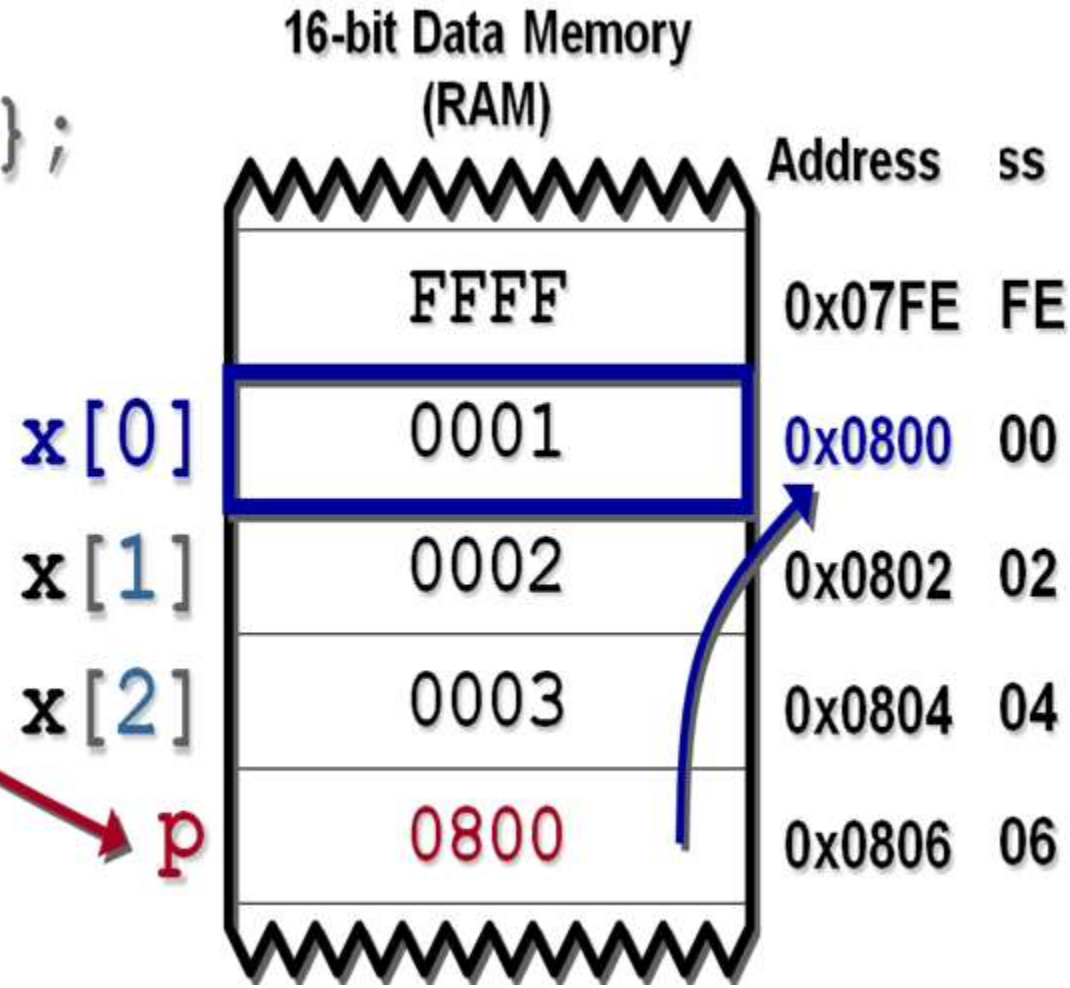
```c
int x[3] = {1,2,3};
int *p;

p = x;
p++;
```



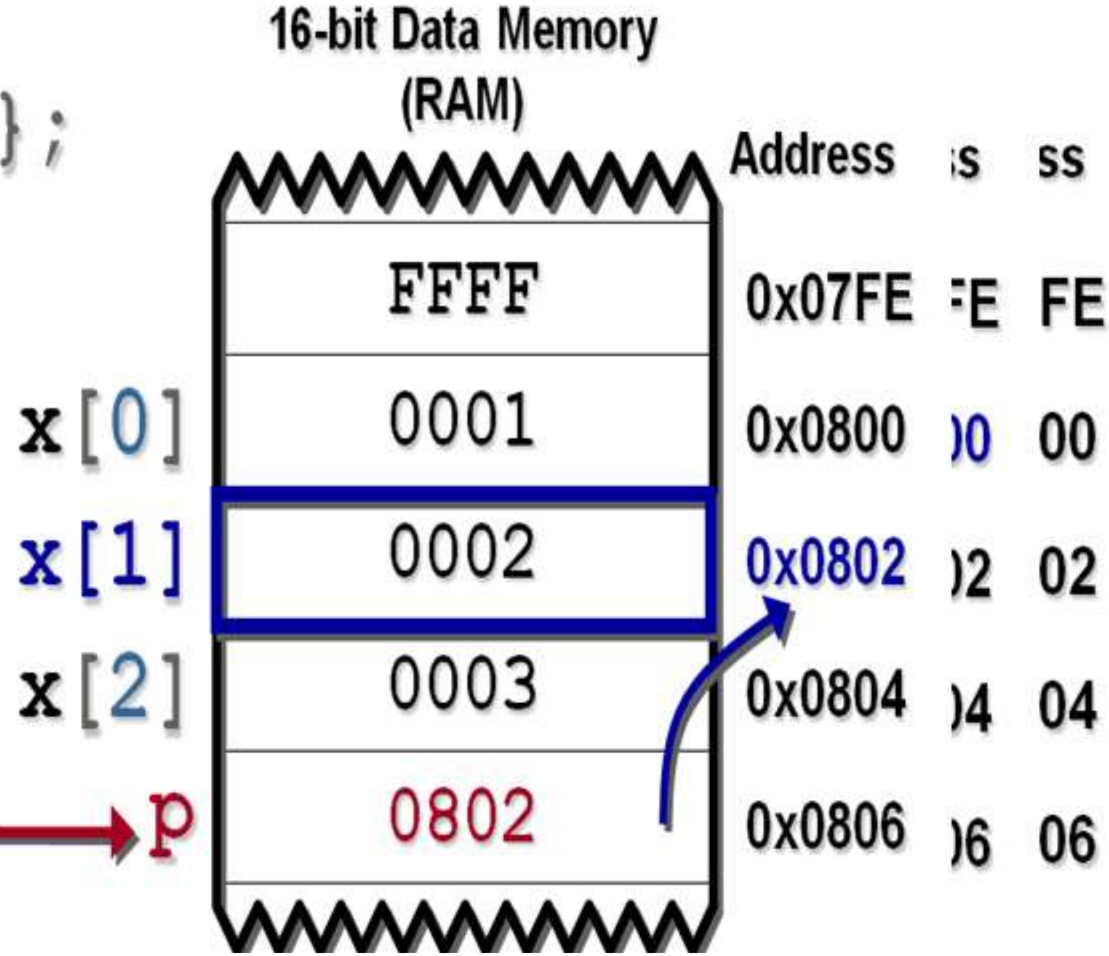
```
int x[3] = {1,2,3};
int *p;

p =  x;
p++;
```



```c
int x[3] = {1,2,3};
int *p;

p = &x[0];
p++;
```



Why Do Pointer Arithmetic?

- May be slightly more efficient.
  - Fewer instructions executed than stepping through an array with an integer index.
  - Normally not significant.
- Slightly more compact notation.
- Closer to programming in machine language.
  - Traditional C culture.

Pointer Arithmetic

- Pointers can be increment only by *integer* values.

- For example, it doesn’t make sense to add two pointers or multiply them.

Attempt to add two pointers

```
main ( )
{
    int some_numbers[] = {101, 102, 103, 104, 105};
    int i;

    int* pNumber = some_numbers;
    int* pNumber2 = &some_numbers[2];

    for (i = 0; i < 3; i++)
    {
       cout << "Entry " << i+2 << " is " << *(pNumber2 + i);

    }
    // pNumber += pNumber2;
    return 0;
}
```

This gets a compile error

Equivalent loops

```
main ( )
{
    int  nums[] = {101, 102, 103, 104, 105};
    int* p_nums = nums;
    for (int i = 0; i < 5; i++)
          cout << nums[i] << " ";

    for (int i = 0; i < 5; i++)
          cout << p_nums[i] << " ";

    for (int i = 0; i < 5; i++)
          cout << *(p_nums + i) << " ";

    for (int i = 0; i < 5; i++)
          cout << *(p_nums++) << " ";    // Careful

   return 0;
}
```

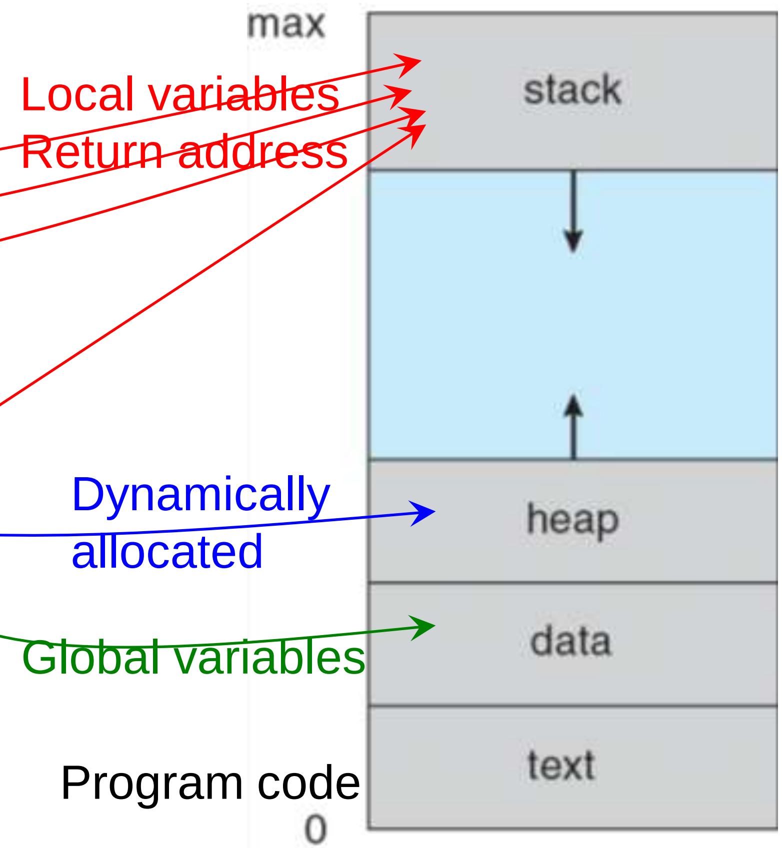
3. Dynamic Allocation

- Static memory allocation
  - Memory space is known and decided and catered for in program memory space (data section) at compile time
- Automatic memory allocation
  - Memory space for local variables in a function is allocated when a function is called (in stack)
- Dynamic memory allocation
  - Memory is allocated as needed during runtime (in heap)

Program (Process) in Memory

```
int global = 0;
int main (int arg)
{
      float  local;
      char *ptr;

      ptr = new int (100);
      local = 0;
      local += 10*5;
      …..
      ….
      foo();
      …. /* return addr */
      ….
      return 0;
}
```



`new` and `delete` Operators

- You can create memory as needed during runtime using **`new`** operator.
- The *lvalue* of the assignment must be a pointer.
  - ***pointer* = new *type;***
  - ***pointer* = new *type* [*size*];**

  - ***delete pointer;***
  - ***delete[] pointer;***

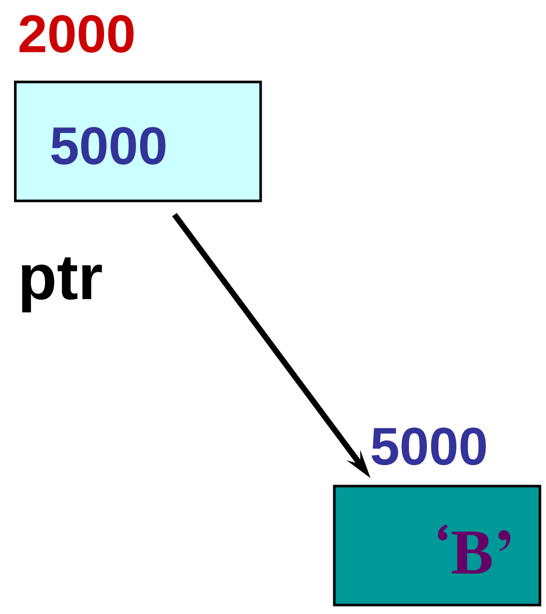
Operator *new*

```
char*  ptr;

ptr = new char;

*ptr = 'B';

cout <<  *ptr;
```



- NOTE: Dynamic data has no variable name

Operator **delete**

```
char*  ptr;

ptr = new char;

*ptr = 'B';

cout  <<  *ptr;

delete  ptr;
```

2000

???

ptr

NOTE:

delete deallocates the memory pointed to by ptr

`new` and `delete` Operators

- `int *arr = new int[45];`
- `...`
- `delete[] arr;`
- **Memory leak** occurs if memory is allcoated but not freed.
- **LAS** London Ambulance System story.

Exercise

- **D**raw diagrams showing the contents of memory after each line of the following code

- `v1 = 10; v2 = 25; p1 = &v1; p2 = &v2;`
- `*p1 += *p2;`
- `p2 = p1;`
- `*p2 = *p1 + *p2;`

4. C++11 Smart Pointers

- Smart pointers point to objects, and when the pointer goes out of scope, the object
- gets destroyed. This makes them *smart* in the sense that we do not have to worry about
- manual deallocation of allocated memory. The smart pointers do all the heavy lifting
- for us.
- There are two kinds of smart pointers, the unique pointer with an *std::unique_*
- *ptr*<*type*> signature and a shared pointer

4. C++11 Smart Pointers

- Smart pointers point to objects, and when the pointer goes out of scope, the objectgets destroyed. This makes them *smart* in the sense that we do not have to worry about manual deallocation of allocated memory.
- There are two kinds of smart pointers:
- **`std::unique_ptr<type>`**
- **`std::shared_ptr<type>`**

4. C++11 Smart Pointers

- We have only one unique pointer pointing at the object.
- In contrast, we can have multiple shared pointers pointing at an object.
- When the unique pointer goes out of scope, the object gets destroyed, and the memory is deallocated.
- When the last of the shared pointers pointing at our object goes out of scope, the object gets destroyed and memory deallocated.

4. C++11 Smart Pointers

```
#include <iostream>
#include <memory>
using namespace std;
int main() {
    unique_ptr<int> p(new int{123});
    cout << *p;
} // p goes out of scope and memory deallocated
  // No need to write delete p;
  // WRONG unique_ptr<int> q = p;
```

4. C++11 Smart Pointers

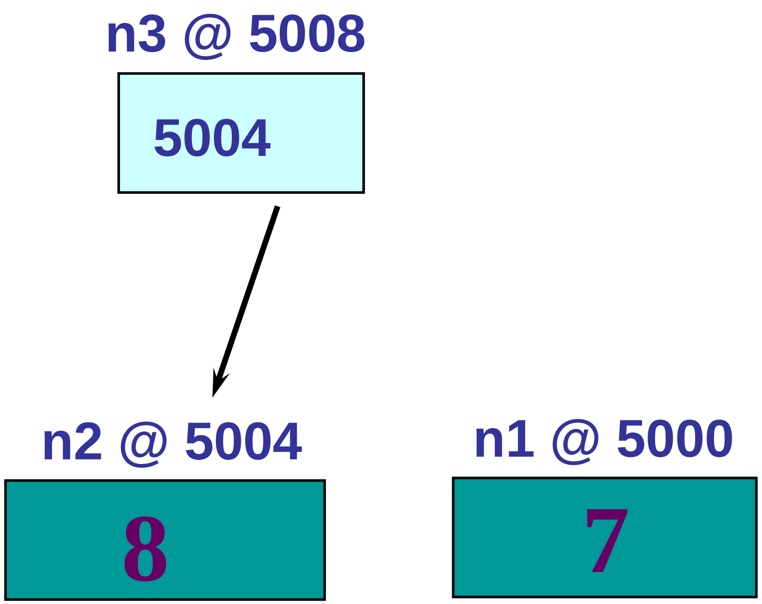
```
#include <iostream>
#include <memory>
using namespace std;
int main() {
    shared_ptr<int> p2(new int{123456});
    cout << *p2 << endl;
    shared_ptr<int> p3(p2); // = p2
    cout << *p3 << endl;
}
```

5. Reference Variable

- A reference variable is an *alias* (another name) to another variable. A reference variable is declared by appending the & sign to a data type.
- It must be initialized when created.
- **`int n1 = 7, n2 = 8;`**
- **`int& n3 = n2; // Now n3 is same as n2`**
- **`n3 = 100;        // n3 = n2 = 100`**
- **`n3 = n1          // n3 = n2 = n1 = 7`**

5. Reference Variable

- `int n1 = 7, n2 = 8;`
- `int& n3 = n2; // Now n3 is same as n2`
- `n3 = 100;      // n3 = n2 = 100`
- `n3 = n1       // n3 = n2 = n1 = 7`



Readings

- Modern C++ for Absolute beginners: Chap 1-22, 27-29
- Pb Solving w C++ (Savitch, 10th ed): Ch 1-5, 7-9 and 14
- Dr Amin Allam Notes on Pointers

Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-1 CS112
2017/2018 2nd Semester

Pointers and references

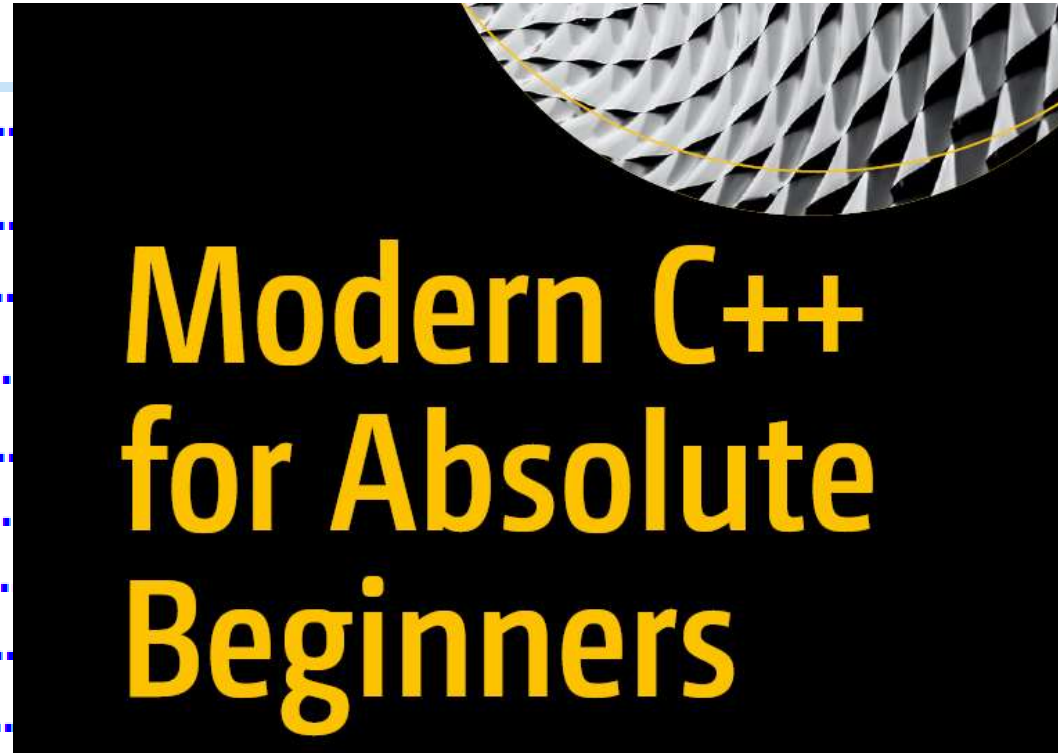
Prepared by:
Dr. Amin Allam

1 References

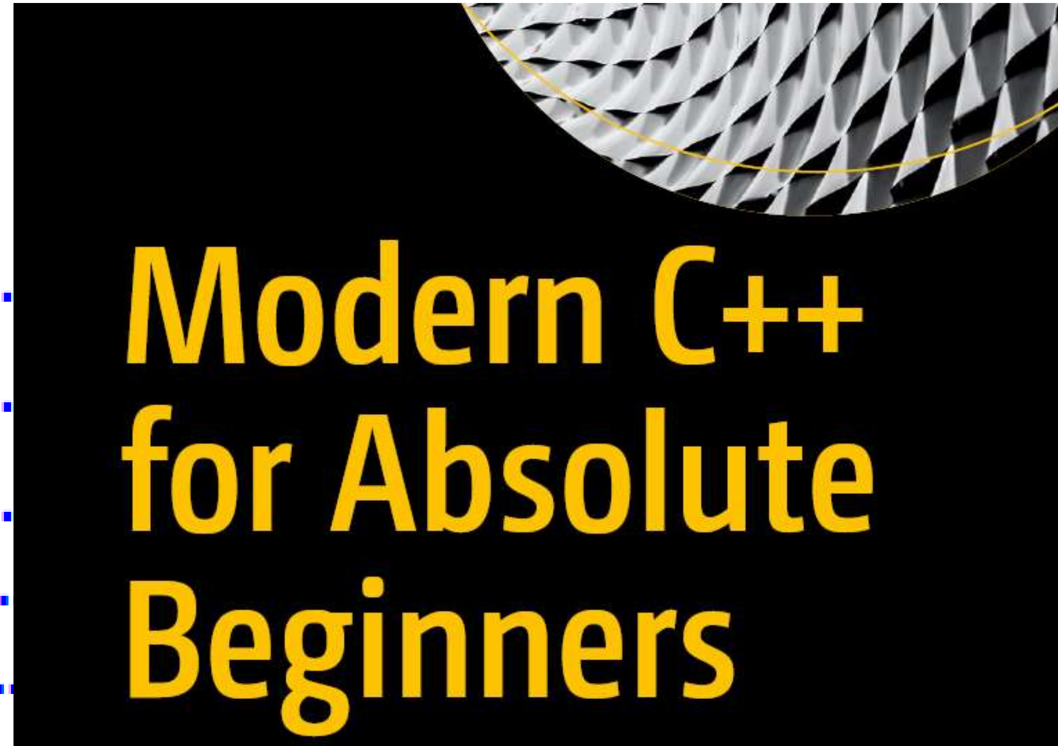
A reference variable is an *alias* (another name) to another variable. A reference variable is declared by appending the & sign to a data type. For example:

```
int t=7, r=8;
int& q=t;   // q is another name for t, any change in q affects t and vice versa
q=6;  cout<<q<<" "<<t<<endl;   // Prints 6 6
t=4;  cout<<q<<" "<<t<<endl;   // Prints 4 4
q=r;  cout<<q<<" "<<t<<" "<<r<<endl;   // Prints 8 8 8
q=3;  cout<<q<<" "<<t<<" "<<r<<endl;   // Prints 3 3 8
```

Readings



Readings



Other Resources

- http://msdn.microsoft.com/en-us/library/3bstk3k5(v=vs.80)
- www.cplusplus.com

أقوال مأثورة

سئل أحد الصالحين من تعز من الناس؟
قال : من أخلاقه كريمة
ومجالسته غنيمة
ونيته سليمة
ومفارقته أليمة
كالمسك كلما مر عليه الزمان
زاد قيمة