

C++ Programming I

Pointers and References

C++ Programming
March 8, 2018

Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- ▶ **Variables and Memory**
- ▶ **Pointers**
- ▶ **Memory Allocation**
- ▶ **References**
- ▶ **Pointers and Arrays**

Lecture 4

Dr. P. Arnold



Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

Variables and Memory

Lecture 4

Dr. P. Arnold



Variables and
Memory

Pointers

Memory Allocation

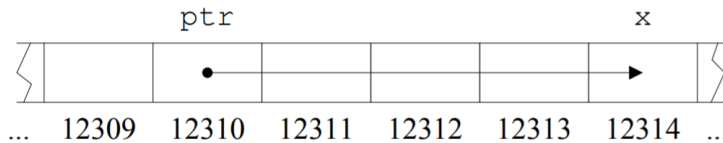
References

Pointers and
Arrays

Variables and Memory

Background I

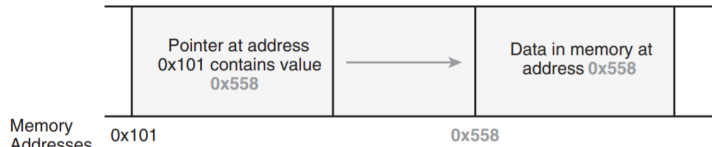
- ▶ When you define a variable, the computer associates the variable name with a particular location in memory and stores a value there. When you refer to the variable by name in your code, the computer must take two steps:
 1. Look up the address that the variable name corresponds to
 2. Go to that location in memory and retrieve or set the value it contains
- ▶ C++ allows us to perform either one of these steps independently on a variable with the `&` and `*` operators:
 1. `&x` evaluates to the address of `x` in memory.
 2. `* (&x)` takes the address of `x` and dereferences it - it retrieves the value at that location in memory. `* (&x)` thus evaluates to the same thing as `x`



Variables and Memory

Background II

- ▶ A pointer is also a variable storing an **address** in memory. Just the same way as a variable of type `int` is used to contain an integer value, a pointer variable is used to contain a memory address



- ▶ A pointer occupies space in memory, e.g. `0x101`, equal to a `int` variable
- ▶ The value contained in a pointer, e.g. `0x558` is interpreted as a memory address
- ▶ A pointer is a special variable that points to a location in memory
- ▶ A pointer that stores the address of some variable `x` is said to **point to** `x`

Pointers

Lecture 4

Dr. P. Arnold



Bern University
of Applied Sciences

Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

Pointers

Usage & syntax I

```
1 // Define variable
2 int value = 23;
3
4 // Declare Pointer
5 int* valuePtr; // very bad!
6
7 // Set valuePtr to not existing address 0
8 int* valuePtr = nullptr;
9
10 // Get adress of value with address operator &
11 int* valuePtr = &value;
12
13 // Get value with indirection Operator *
14 int copyValue = *valuePtr;
```

► Never declare a pointer without initialisation!

C is not C++

Use `nullptr` and not C-Style `NULL` macro

Pointers

Usage & syntax II

```
1 int age = 30;
2 int dogsAge = 9;
3
4 cout << "Integer age = " << age << endl;
5 // --> Integer age = 30
6 cout << "Integer dogsAge = " << dogsAge << endl;
7 // --> Integer dogsAge = 9
8
9 int* intPtr = &age;
10
11 // Displaying the value of pointer
12 cout << "intPtr = 0x" << hex << intPtr << endl;
13 // intPtr = 0x0025F788
14
15 // Displaying the value at the pointed location
16 cout << "*intPtr = " << dec << *intPtr << endl;
17 // *intPtr = 30
18
19 intPtr = &dogsAge;
20
21 cout << "intPtr = 0x" << hex << intPtr << endl;
22 // intPtr = 0x0025F77C
23
24 cout << "*intPtr = " << dec << *intPtr << endl;
25 // *intPtr = 9
```

Lecture 4

Dr. P. Arnold



Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

Pointers

Size of Pointer

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "sizeof fundamental types -" << endl;
7     cout << "sizeof(char) = " << sizeof(char) << endl;
8     cout << "sizeof(int) = " << sizeof(int) << endl;
9     cout << "sizeof(double) = " << sizeof(double) <<
10         endl;
11
12     cout << "sizeof pointers to fundamental types -" <<
13         endl;
14     cout << "sizeof(char*) = " << sizeof(char*) << endl;
15     cout << "sizeof(int*) = " << sizeof(int*) << endl;
16     cout << "sizeof(double*) = " << sizeof(double*) <<
17         endl;
18
19     return 0;
20 }
```

Output:

```
1 sizeof fundamental types -
2 sizeof(char) = 1
3 sizeof(int) = 4
4 sizeof(double) = 8
5 sizeof pointers to fundamental types -
6 sizeof(char*) = 4
7 sizeof(int*) = 4
8 sizeof(double*) = 4
```

Pointers

As Function Argument

- ▶ Just like any other data type, we can pass pointers as arguments to functions. The same way we'd say `void func(int x) ...`, we can say `void func(int *x) ...`

```
1 void squareByPtr(int* numPtr)
2 {
3     *numPtr = *numPtr * *numPtr ;
4 }
5
6 int main()
7 {
8     int x = 5;
9     squareByPtr(&x);
10    cout << x; // Prints 25
11 }
```

```
1 void squareByPtr(int& numPtr)
2 {
3     numPtr = numPtr * numPtr ;
4 }
5
6 int main()
7 {
8     int x = 5;
9     squareByPtr(x);
10    cout << x; // Prints 25
11 }
```

- ▶ Note the varied uses of the `*` operator on line 3 (readability!)
- ▶ The use of pointers as Parameters is inconvenient and error-prone

C is not C++

Prefer references to pointers

Pointers

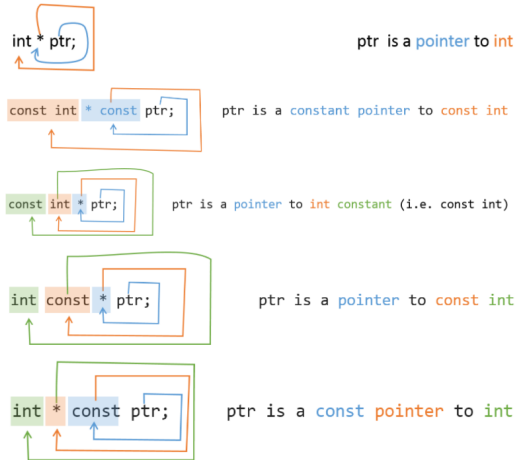
Const Pointers

- ▶ There are two places the `const` keyword can be placed within a pointer variable declaration. This is because there are two different variables whose values you might want to forbid changing: the pointer itself and the value it points to.
- 1. `const int * ptr;`
Declares a changeable pointer to a **constant integer**. The integer value cannot be changed through this pointer, but the pointer may be changed to point to a different constant integer.
- 2. `int * const ptr;`
Declares a **constant pointer** to changeable integer data. The integer value can be changed through this pointer, but the pointer may not be changed to point to a different constant integer.
- 3. `const int * const ptr;`
Declares a **constant pointer** to a **const integer**

Pointers

Const Pointers - Clockwise/Spiral Rule

- ▶ `int *` - pointer to `int`
- ▶ `int const *` - pointer to const int
- ▶ `int * const` - const pointer to `int`
- ▶ `int const * const` - const pointer to const int



Memory Allocation

Lecture 4

Dr. P. Arnold



Bern University
of Applied Sciences

Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

Memory Allocation

C++ supports three basic types of memory allocation:

1. **Static memory** allocation happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
 2. **Automatic memory** allocation happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited
 3. **Dynamic memory** allocation is done by the programmer!
- Both static and automatic allocation have two things in common:
1. The size of the variable / array must be known at compile time
 2. Memory allocation and deallocation happens automatically (when the variable is instantiated / destroyed).

```
1 int globalVar = 100; // static memory
2
3 int main()
4 {
5     if(true)
6     {
7         int autoVar = 23; // automatic memory
8     } // autoVar freed
9
10    int* dynArray = new int[100]; // dynamic array
11    delete [] dynArray; // free manually
12    return 0;
13 } // globalVar freed
```

Memory Allocation

Limits of Static and Automatic Memory Allocation

- ▶ If we have to declare the size of everything at compile time!

```
1 // let's hope
2 char name[25]; // their name is less than 25 chars!
3 Record record[500]; // there are less than 500
   records!
4 Monster monster[40]; // 40 monsters is maximum
5 Polygon rendering[30000]; // 30'000 polygons are
   enough
```

- ▶ This is a poor solution for many reasons:
 1. It leads to wasted memory if the variables aren't actually used
 2. Most normal variables (including fixed arrays) are allocated in a portion of memory called the stack. The amount of stack memory for a program is generally quite small – Visual Studio defaults the stack size to 1MB. If you exceed this number, **stack overflow** will result, and the operating system will probably close down the program
 3. Being limited to just 1MB of memory would be problematic for many programs

Note:

Run-time allocation of static arrays might be supported by our compiler (C99)

Memory Allocation

Stack Overflow

- ▶ Most importantly, it can lead to artificial limitations and/or array overflows

```
1 int main()
2 {
3     int array[1000000]; // allocate 1 million
                          // integers (probably 4MB of memory)
4 }
```

- ▶ Fortunately, these problems are easily addressed via **dynamic memory allocation**
 - ▶ Dynamic memory allocation is a way for running programs to request memory from the operating system when needed
 - ▶ This memory does not come from the program's limited stack memory – instead, it is allocated from a much larger pool of memory managed by the operating system called the heap
 - ▶ On modern machines, the heap can be gigabytes in size

Lecture 4

Dr. P. Arnold



Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

Dynamic Memory Allocation

Keywords `new` and `delete`

- ▶ C++ supplies you two operators, `new` and `delete`, for the management of the memory consumption of your application

```
1 Type* ptr = new Type; // allocate memory
2 delete ptr; // release memory allocated above
3
4 Type* ptr = new Type[numElements]; // allocate a
   block
5 delete[] ptr; // release block allocated above
```

- ▶ Allocate objects with `new` **and** free with `delete`
- ▶ When declaring an array, the array **is** a pointer to the first element!
- ▶ Free arrays with `delete[]`

C++ is not C

Use `new` and `delete` and not C-style `malloc` and `free`

Lecture 4

Dr. P. Arnold



Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

Pointers

Common Pitfalls - Memory Leaks

► Memory Leaks

```
1 // initial allocation
2 int* intPtr = new int[5];
3
4 // use intPtr
5 ...
6 // forget to release using delete[] intPtr;
7 ...
8 // make another allocation and overwrite
9 intPtr = new int[10]; // leaks the previously allocated
                        // memory!
```

Pointers

Common Pitfalls - Non Valid Memory Location

► Non-valid memory locations

```
1 // uninitialized pointer (bad!)
2 int* badPtr;
3
4 // badPtr contains garbage value, i.e. not initialized!
5 cout << "Value of badPtr: " << *badPtr << endl;
6
7 // -> crash or undefined behaviour
```

► At least assign nullptr:

```
1 // at least do:
2 int* badPtr = nullptr;
3
4 // Explicitly check for validity every time!
5 if(badPtr != nullptr)
6 {
7     cout << "Value of badPtr: " << *badPtr << endl;
8 }
```

Pointers

new(nothrow)

- Check success of `new` without using exception handling

```
1 // Request LOTS of memory space, use nothrow
2 int* ptr = new(nothrow) int[0xffffffff];
3
4 if(ptr != nullptr) // check ptr != nullptr
5 {
6     // Use the allocated memory
7     delete[] ptr;
8 }
9 else
10 {
11     cout << "Memory allocation failed. Ending
12         program" << endl;
```

- `new(nothrow)` returns a `nullptr` on failure

Note:

Add `#include <new>` to use `std::nothrow`

Pointers

Do I absolutely need Dynamic Memory Allocation?

- ▶ Always think about using an already implemented, efficient and tested standard container from the standard template library (STL), e.g `vector`
- ▶ If you need pointers, use **smart pointers**
- ▶ If for some reasons you have to work with pointers follow the best practice rules to make live easier.

DO	DON'T
<p>DO always initialize pointer variables, or else they will contain junk values. These junk values are interpreted as address locations—ones your application is not authorized to access. If you cannot initialize a pointer to a valid address returned by <code>new</code> during variable declaration, initialize to <code>NULL</code>.</p> <p>DO ensure that your application is programmed in a way that pointers are used when their validity is assured, or else your program might encounter a crash.</p> <p>DO remember to release memory allocated using <code>new</code> by using <code>delete</code>, or else your application will leak memory and reduce system performance.</p>	<p>DON'T access a block of memory or use a pointer after it has been released using <code>delete</code>.</p> <p>DON'T invoke <code>delete</code> on a memory address more than once.</p> <p>DON'T leak memory by forgetting to invoke <code>delete</code> when done using an allocated block of memory.</p>

References

Lecture 4

Dr. P. Arnold



Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

References

References vs Pointers

- ▶ When we write `void f(int &x) ...` and call `f(y)`, the reference variable `x` becomes another name - an alias - for the value of `y` in memory. We can declare a reference variable locally, as well:

```
1 int y;  
2 int &x = y; // Makes x a reference to, or alias of, y
```

- ▶ References are similar to pointers but are dereferenced every time they are used. The only differences between using pointers and using references are:
 - ▶ References are sort of **pre-dereferenced** – you do not dereference them explicitly
 - ▶ You cannot change the location to which a reference points, whereas you can change the location to which a pointer points. Because of this, **references must always be initialized** when they are declared
 - ▶ `nullptr` references are not possible

References

as Parameters in Functions

- References are very useful when programming functions

```
1 #include <iostream>
2 using namespace std;
3
4 void GetSquare(int number) // call by copy
5 {
6     number *= number;
7 }
8
9 int main()
10 {
11     cout << "Enter a number you wish to square: ";
12     int number = 0;
13     cin >> number;
14
15     GetSquare(number);
16     cout << "Square is: " << number << endl;
17
18     return 0;
19 }
```

```
1 #include <iostream>
2 using namespace std;
3
4 void GetSquare(int& number) // call by reference
5 {
6     number *= number;
7 }
8
9 int main()
10 {
11     cout << "Enter a number you wish to square: ";
12     int number = 0;
13     cin >> number;
14
15     GetSquare(number);
16     cout << "Square is: " << number << endl;
17 }
```


Pointers and References

Different Faces of * and &

- ▶ The usage of the * and & operators with pointers/references can be confusing. The * operator is used in two different ways:
 1. When **declaring a pointer**, * is placed before the variable name to indicate that the variable being declared is a pointer and not a value
 2. When using a pointer that has been set to point to some value, * is placed before the pointer name to **dereference** it - to access or set the value it points to (**indirection operator**)

- ▶ A similar distinction exists for &, which can be used either:
 1. to indicate a **reference data type** (as in `int &x;`), or
 2. to take the **address of** a variable (as in `int *ptr = &x;`).

Pointers and Arrays

Lecture 4

Dr. P. Arnold



Bern University
of Applied Sciences

Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

Pointers and Arrays

Arrays are Pointers, that means:

1. When declaring an array `int a[n]`, the array, i.e. `a` is of type `int*` `a` and the value of `a` is the address of the first element
2. When dereferencing the array with `*a` one gets the value of the first element.

► `a` is a pointer and not the full array - `a[0]` and `*a` are equivalent

```
1 int* a = new int[n]; // a is adress of first element
2
3 int firstValue = *a; // is the value of the first element
4
5 // Number elements
6 int aSize = sizeof(a)/sizeof(*a);
```

Pointers and Arrays

Array-Pointer Dualism

```
1 // Static Allocation – Size set at compile time
2 //
3
4 int x[10] = {0}; // 10 elements of type int initialized
   to 0
5 int first = x[0]; // indexing first element
6 int last = x[9]; // indexing last element
7
8 int x[n]; // Compile error – size must be known
9
10 // Dynamic Allocation – Size set at runtime
11 //
12
13 int* x[10] = new int[n];
14
15 // Access – Arrays are pointers!
16 //
17
18 int x = *(a+index); // pointer access is the same as
19 int x = a[index]; // index access
```

- ▶ The name of an array is actually a pointer to the first element in the array
- ▶ Array indices start at 0: the first element of an array is the element that is 0 away from the start of the array

Pointers and Arrays

Pointer Arithmetic

```
1 // C-Style Pointer Arithmetic
2 int sum(const int *p, int n)
3 {
4     int i, x;
5     for (x=0, i=0; i<n; i++) x += *p++;
6     return(x);
7 }
```

- ▶ The operators +=, -=, ++, -- are allowed on non-const pointers
- ▶ p++ means that pointer p points to the next value after evaluation

Lecture 4

Dr. P. Arnold



Variables and
Memory

Pointers

Memory Allocation

References

Pointers and
Arrays

Thank You
Questions

???

Lecture 4

Dr. P. Arnold



Variables and
Memory

Pointers

Memory Allocation

References

**Pointers and
Arrays**