C++ Programming I

Pointers and References

C++ Programming
FS 2020

Dr. P. Arnold &
Dr. A. Schneider
Bern University of Applied Sciences

Agenda

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

Variables and Memory

Variables and Pointers in Memory
Background

```
// Variable
int var = 23;

Memory address:   ...    1001    1002    1003    1004   1005 ...
                 ---------------------------------------------
Stored value:    |      |       |       |  23   |      |      |
                 ---------------------------------------------
Variable name:                             var
```

Variables and Pointers in Memory
Background

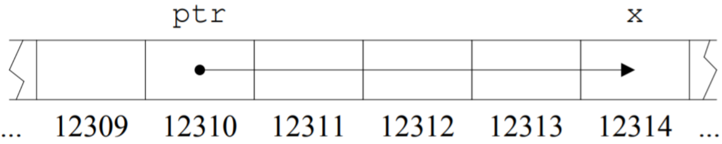
```
// Variable
int var = 23;

Memory address:   ...    1001    1002    1003    1004   1005 ...
                 ---------------------------------------------
Stored value:    |      |       |       |  23   |      |      |
                 ---------------------------------------------
Variable name:                            var

// Pointer to variable
int* varPtr = &var;


Memory address:   ...    1001    1002    1003    1004   1005 ...
                 ---------------------------------------------
Stored value:    |      | 1003  |       |  23   |      |      |
                 ---------------------------------------------
Variable name:           varPtr           var
```

- Reference operator `&` gets address of a variable: `&var -> 1003`
- `varPtr` is a pointer to an integer value (type: `int*`)
- Dereference operator `*` gets value pointed by a value: `*valPtr -> 23`
- Hence, `*(&var) -> 23`

Variables and Pointers in Memory

Background I

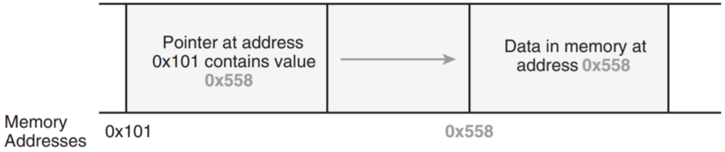
- When you define a variable, the computer associates the variable name with a particular location in memory and stores a value there. When you refer to the variable by name in your code, the computer must take two steps:
    1. Look up the address that the variable name corresponds to
    2. Go to that 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 Pointers in Memory

Background II

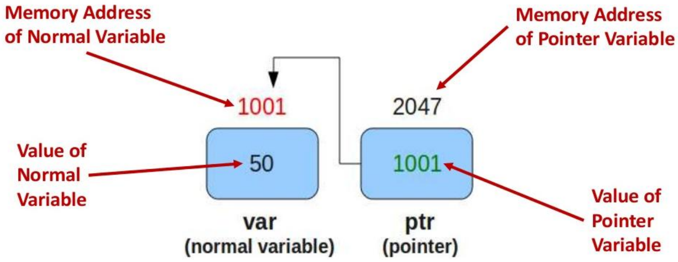
- A pointer is also a variable storing an **address** in memory. Just the same way as a variable of type `int` is used to contain an integer value, a pointer variable is used to contain a memory address



- 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`

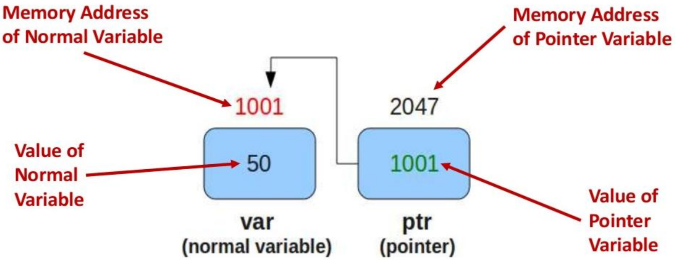
Variables and Pointers in Memory

Background III:



Variables and Pointers in Memory

Background III:



- `int var = 50;`
- `int* ptr = &var;`
- Referencing var: `&var -> 1001`
- Dereferencing ptr: `*ptr -> 50`
- Referencing ptr: `&ptr -> 2047`

Pointers

Pointers

Usage & syntax I

```
// Define variable
int value = 23;

// Declare Pointer
int* valuePtr; // very bad!

// Set valuePtr to not existing address 0
int* valuePtr = nullptr;

// Get adress of value with address operator &
int* valuePtr = &value;

// Get value with indirection Operator *
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

```
int age = 30;
int dogsAge = 9;

cout << "Integer age = " << age << endl;
// --> Integer age = 30
cout << "Integer dogsAge = " << dogsAge << endl;
// --> Integer dogsAge = 9

int* intPtr = &age;

// Displaying the value of pointer
cout << "intPtr = 0x" << hex << intPtr << endl;
// intPtr = 0x0025F788

// Displaying the value at the pointed location
cout << "*intPtr = " << dec << *intPtr << endl;
// *intPtr = 30

intPtr = &dogsAge;

cout << "intPtr = 0x" << hex << intPtr << endl;
// intPtr = 0x0025F77C

cout << "*intPtr = " << dec << *intPtr << endl;
// *intPtr = 9
```

Pointers

Size of Pointer

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

int main()
{
    cout << "sizeof fundamental types -" << endl;
    cout << "sizeof(char) = " << sizeof(char) << endl;
    cout << "sizeof(int) = " << sizeof(int) << endl;
    cout << "sizeof(double) = " << sizeof(double) << endl;

    cout << "sizeof pointers to fundamental types -" << endl;
    cout << "sizeof(char*) = " << sizeof(char*) << endl;
    cout << "sizeof(int*) = " << sizeof(int*) << endl;
    cout << "sizeof(double*) = " << sizeof(double*) << endl;

    return 0;
}
```

Pointers

Size of Pointer

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

int main()
{
    cout << "sizeof fundamental types -" << endl;
    cout << "sizeof(char) = " << sizeof(char) << endl;
    cout << "sizeof(int) = " << sizeof(int) << endl;
    cout << "sizeof(double) = " << sizeof(double) << endl;

    cout << "sizeof pointers to fundamental types -" << endl;
    cout << "sizeof(char*) = " << sizeof(char*) << endl;
    cout << "sizeof(int*) = " << sizeof(int*) << endl;
    cout << "sizeof(double*) = " << sizeof(double*) << endl;

    return 0;
}
```

Output:

```
sizeof fundamental types -
sizeof(char) = 1
sizeof(int) = 4
sizeof(double) = 8
sizeof pointers to fundamental types -
sizeof(char*) = 4
sizeof(int*) = 4
sizeof(double*) = 4
```

32 bit, size of pointer does not depend on var type, but on ram

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 }
```

- 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

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 }
```

- 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

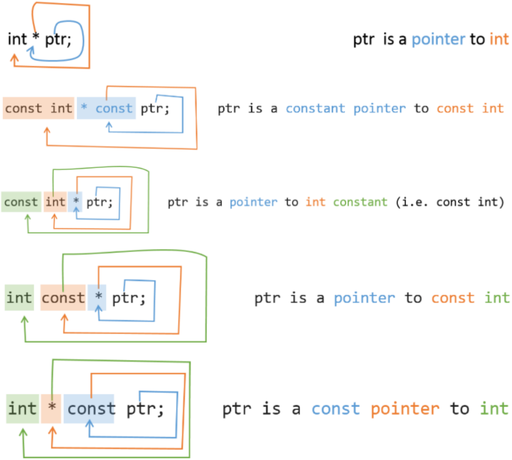
- There are two places the `const` keyword can be placed within a pointer variable declaration. This is because there are two different variables whose values you might want to forbid changing: the pointer itself and the value it points to.

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

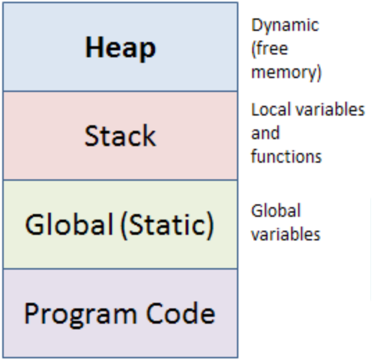


Memory Allocation

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!



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).

```cpp
int globalVar = 100; // static memory

int main()
{
    if(true)
    {
        int autoVar = 23; // automatic memory
    } // autoVar freed

    int* dynArray = new int[100]; // dynamic array
    delete [] dynArray; // free manually
    return 0;
} // globalVar freed
```

Memory Allocation

Limits of Static and Automatic Memory Allocation

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

```
// let's hope
char name[25]; // their name is less than 25 chars!
Record record[500]; // there are less than 500 records!
Monster monster[40]; // 40 monsters is maximum
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

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

- 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

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

```
Type* ptr = new Type; // allocate memory
delete ptr; // release memory allocated above

Type* ptr = new Type[numElements]; // allocate a block
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`

Pointers
Common Pitfalls - Memory Leaks

- Memory Leaks

```
// initial allocation
int* intPtr = new int[5];

// use intPtr
...
// forget to release using delete[] intPtr;
...
// make another allocation and overwrite
intPtr = new int[10]; // leaks the previously allocated memory!
```

doesnt use the previous allocated memory

Pointers
Common Pitfalls - Non Valid Memory Location

- Non-valid memory locations

```cpp
// uninitialized pointer (bad!)
int* badPtr;

// badPtr contains garbage value, i.e. not initialized!
cout << "Value of badPtr: " << *badPtr << endl;

// -> crash or undefined behaviour
```

Pointers

Common Pitfalls - Non Valid Memory Location

- Non-valid memory locations

```cpp
// uninitialized pointer (bad!)
int* badPtr;

// badPtr contains garbage value, i.e. not initialized!
cout << "Value of badPtr: " << *badPtr << endl;

// -> crash or undefined behaviour
```

- At least assign `nullptr`:

```cpp
// at least do:
int* badPtr = nullptr;

// Explicitly check for validity every time!
if(badPtr != nullptr)
{
    cout << "Value of badPtr: " << *badPtr << endl;
}
```

Pointers

`new(nothrow)`

- Check success of `new` without using exception handling

```
// Request LOTS of memory space, use nothrow
int* ptr = new(nothrow) int[0x1fffffff];

if(ptr != nullptr) // check ptr != nullptr
{
    // Use the allocated memory
    delete[] ptr;
}
else
{
    cout << "Memory allocation failed. Ending 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
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 `new` during variable declaration, initialize to `NULL`.	**DON'T** access a block of memory or use a pointer after it has been released using `delete`.
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.	**DON'T** invoke `delete` on a memory address more than once.
DO remember to release memory allocated using `new` by using `delete`, or else your application will leak memory and reduce system performance.	**DON'T** leak memory by forgetting to invoke `delete` when done using an allocated block of memory.

References

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:

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

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:

```
int y;
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

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

void getSquare(int number) // call by copy
{
    number *= number;
}

int main()
{
    cout << "Enter a number you wish to square: ";
    int number = 0;
    cin >> number;

    getSquare(number);
    cout << "Square is: " << number << endl;

    return 0;
}
```

References

as Parameters in Functions

- References are very useful when programming functions

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

void getSquare(int& number) // call by reference
{
    number *= number;
}

int main()
{
    cout << "Enter a number you wish to square: ";
    int number = 0;
    cin >> number;

    getSquare(number);
    cout << "Square is: " << number << endl;

    return 0;
}
```

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

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

```
int* a = new int[n]; // a is adress of first element

int firstValue = *a; // is the value of the first element

// Number elements is unknown!
int aSize = sizeof(a)/sizeof(*a); // aSize = 1!
// Prefer vector or save size
```

Pointers and Arrays

Array-Pointer Dualism

```
// Static Allocation — Size set at compile time
// ————————————————————————————————————————————————

int x[10] = {0};  // 10 elements of type int initialized to 0
int first = x[0]; // indexing first element
int last = x[9];  // indexing last element

int x[n]; // Compile error — size must be known

// Dynamic Allocation — Size set at runtime
// ————————————————————————————————————————————————

int* x = new int[n];

// Access — Arrays are pointers!
// ————————————————————————————————————————————————

int x = *(a+index); // pointer access is the same as
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

```
// C-Style Pointer Arithmetic
int sum(const int *p, int n)
{
   int i,x;
   for (x=0,i=0; i<n; i++) x += *p++;
   return(x);
}
```

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

Thank You

Questions

???