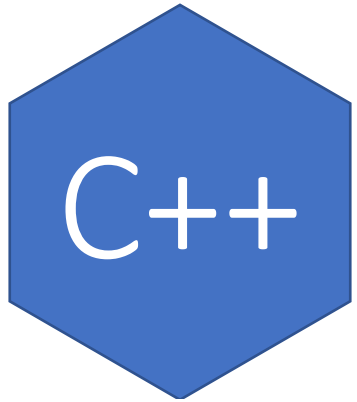
# Computer Programming with





Presentation By AwesomeKen

# Outline


- TYPE Modifiers
- TYPE Qualifiers
- POINTERS
- REFERECES

# Type Modifiers


- From previous lectures we know that C++ is a strongly typed language, ie. For every data we want to store the type of the data must be explicitly defined for storage.
- The various built-in datatypes in C++ were also discussed (int, float, double, char, etc.).
- Each of the various data-types have a maximum range of numbers it can hold depending on the size allocated to the data-type by the compiler.

| Type | Bit Width | Typical Range |
|---|---|---|
| char | 8 | –128 to 127 |
| unsigned char | 8 | 0 to 255 |
| signed char | 8 | –128 to 127 |
| int | 32 | –2,147,483,648 to 2,147,483,647 |
| unsigned int | 32 | –2,147,483,648 to 2,147,483,647 |
| signed int | 32 | 0 to 4,294,967,295 |
| short int | 16 | –32,768 to 32,767 |
| unsigned short int | 16 | –32,768 to 32,767 |
| signed short int | 16 | 0 to 65,535 |
| long int | 32 | Same as **int** |
| signed long int | 32 | Same as **signed int** |
| unsigned long int | 32 | Same as **unsigned int** |
| float | 32 | 1.8E–38 to 3.4E+38 |
| double | 32 | 2.2E–308 to 1.8E+308 |
| long double | 64 | 2.2E–308 to 1.8E+308 |
| bool | N/A | True or false |
| wchar_t | 16 | 0 to 65,535 |

# Type Modifiers


- However this range can be altered depending on the Type Modifier applied to it.
- As seen here char which is 8bit can hold 2^8 = 255; since char is signed ie. Can hold both positive and negative numbers it is divided into two, with -128 to 127.
- All types without an explicit type modifier are by default signed
- Unsinged means the variable will not hold any negative number eg. A variable for age; there will never be a negative age value. Hence in this case the unsigned can beused to modify the type, increasing the range from 0 to 255.

| Type | Bit Width | Typical Range |
|---|---|---|
| char | 8 | –128 to 127 |
| unsigned char | 8 | 0 to 255 |
| signed char | 8 | –128 to 127 |
| int | 32 | –2,147,483,648 to 2,147,483,647 |
| unsigned int | 32 | –2,147,483,648 to 2,147,483,647 |
| signed int | 32 | 0 to 4,294,967,295 |
| short int | 16 | –32,768 to 32,767 |
| unsigned short int | 16 | –32,768 to 32,767 |
| signed short int | 16 | 0 to 65,535 |
| long int | 32 | Same as **int** |
| signed long int | 32 | Same as **signed int** |
| unsigned long int | 32 | Same as **unsigned int** |
| float | 32 | 1.8E–38 to 3.4E+38 |
| double | 32 | 2.2E–308 to 1.8E+308 |
| long double | 64 | 2.2E–308 to 1.8E+308 |
| bool | N/A | True or false |
| wchar_t | 16 | 0 to 65,535 |

# Type Qualifiers


Type qualifiers determine how variables are accessed and modified.
Example of type qualifiers include const, volatile etc.
Const – means the variable is a constant and cannot be changed once the variable is created.
A const variable must be initialized on creation.
A variable is declared const when the contents of the variable are not to be changed.
Const on function parameters – when const is applied to a function parameter, it prevents the value of the parameter from being modified in the function.

const.cpp

```
1   #include <iostream>
2   int const PIE = 3.142;
3
4   int adder(const int, const int);
5   // int changePie();
6   int main() {
7
8       // int a = 2;
9       // int b = 2 *a;
10
11      int c = adder(a, b);
12
13
14      std::cout << "Compiled successfully!!" << std::endl;
15      // std::cout << changePie() << std::endl;
16      return 0;
17  }
18
19  int adder(const int a, const int b) {
20      return (a + b);
21  }
22
23
24
25
```

const parameters prevents any changes to be made to the arguments.
The constness is scoped only within the block of the function.

# Type Qualifiers


Volatile – means the variable will be altered in ways not explicitly defined by the program. Eg.
The variable can be altered by an interrupt clock.

```
8   #define F_CPU 16000000UL;
9   #include <avr/io.h>
10  #include <avr/interrupt.h>
11
12  #define ALARMON PINB |= (1 << PINB1)
13  #define ALARMOFF PINB |= (1 << PINB2)
14  #define SWITCHON !(PINB & (1 << PINB0))
15
16  volatile int MkSec = 0;
17  int main(void)
18  {
19      //DDRB = (1 << DDB1) | (1 << DDB2);
20      DDRB |= (1 << DDB0);
21
22      TCCR0A = (1 << WGM01);
23      OCR0A = 255;
24
25      TIMSK0 = (1 << OCIE0A);
26
27      sei();
28
29      TCCR0B = (1 << CS02);//|(1 << CS00);
30
31
32      /* Replace with your application code */
33      while (1)
34      {
35      }
36  }
37
38  ISR(TIMER0_COMPA_vect){
39      MkSec++;
40      if(MkSec > 250){
41          PORTB ^= (1 << PORTB0);
42          MkSec = 0;
43      }
44
45  }
```

MkSec is changed by an external interrupt service routine.

# Pointers


- As we have discussed in the previous slides all data in the computer is stored in an address in the memory of the computer.
- The name of the variable is just linked to the memory address so that whenever and wherever the name of the variable is called that address is accessed.
- For example,

```
int  year = 2020;
The  compiler finds location in memory say 0xFFBC and places the
value 2020 in that and links the name “year” to it.
So that whenever we try to get the value of year, the compiler
fetches the value at memory address 0xFFBC.
```

  - When an assignment is done ( `int new_year = year;` ) or a variable is passed to a function eg. `make_date(year),` a copy of the value 2020 is made and stored in the new addresses. To avoid this copying you would like to get the memory address which would give exactly where the value is stored.
  - However, the address (which is also a value) has to be stored in a container and such a container is called POINTER.
  - A pointer is a variable that holds the address of another variable

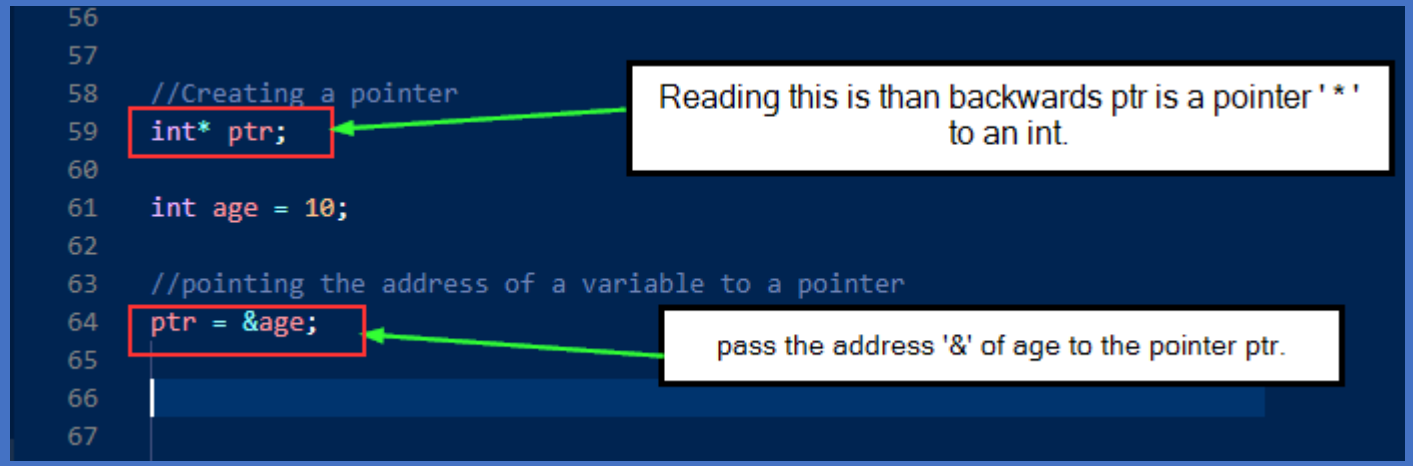
# Pointers


- To create a pointer the * operator is attached to either the beginning of variable name or the end of the data type.

```
int* year; //first instance
Int *year; //second instance
```

- To get the address of the variable the address of operator & is used.

```
int* ptr; //first instance
int year = 2020;
year = &year;
```



# Pointers variables


- Accessing the value  at the address the pointer points to is called DEREFERENCING . This is done with the * operator. Which is called the dereferencing operator.
- If the value at the address the pointer points to is to be changed, the pointer has to be first dereferenced and then the new value assigned to it.
- Eg. `int age = 20;`

```
int* ptr;
ptr = &age;
*ptr = 30;
```

Not age is also 30, since it is the same location which the pointer is pointed to. This is really powerful.

```
58  //Creating a pointer
59  int* ptr;
60
61  int age = 10;
62
63  //pointing the address of a variable to a pointer
64  ptr = &age;
65
66  cout << "age : " << age << endl;
67  cout << "ptr : " << *ptr << endl;
68
69  //Lets change the value of age with ptr
70  cout << "changing the value of age through the pointer " << endl;
71  *ptr = 20;
72
73  cout << "age : " << age << endl;
74  cout << "ptr : " << *ptr << endl;
75
76
77      return 0;
78  }
```

Dereferencing a pointer. Done to get the value at the address being pointed to

```
C:\Users\ROOT\Documents\CodeBlocks\addFunction\bin\Debug\addFu
age : 10
ptr : 10
changing the value of age through the pointer
age : 20
ptr : 20
```

Result

# Pointers variables


- Pointers are also used to get memory dynamically from the **heap**.
- There are two kind of memories used by computer programs they are the ***Stack*** and the ***Heap***.
- Stack memory is allocated to the program at compile time. Therefore the program must know before hand the amount of memory it requires before compilation. Once the memory has been allocated it cannot be changed.
- Heap memory is allocated during runtime, hence there is that flexibility to appropriately select the right amount of memory the program needs as of the time the program is running. Powerful right!

```
51
52      int *ptr = new int;
53
54      //freeing memory
55      delete ptr;
56
57      ptr = NULL;
58
```

```
52      int *ptr;
53      ptr = new int;
54
55      //freeing memory
56      delete ptr;
57
58      ptr = NULL;
59
```

# Pointers variables


- But with great power comes great responsibility. Since you allocated the memory, you are also responsible to deallocate it. Without deallocating such memory the program still holds that memory even when the memory is no longer in use. This is called Memory Leak.
- To allocate memory in C++ the *new* keyword is used and to deallocate memory the *delete* keyword is used.
- Format :

    data-type* var-name = *new* data-type; //initializing a pointer
    data-type* var-name; //declaring a pointer
    var-name = *new* data-type; //later allocating memory

To release/ free the memory :

    *delete* var-name;

```
51
52      int *ptr = new int;
53
54      //freeing memory
55      delete ptr;
56
57      ptr = NULL;
58
```

```
52      int *ptr;
53      ptr = new int;
54
55      //freeing memory
56      delete ptr;
57
58      ptr = NULL;
59
```

# Pointers variables – Dynamic Allocation


- To allocate memory in C++ the *new* keyword is used and to deallocate memory the *delete* keyword is used.
- Format :

  //initializing a pointer
  data-type* var-name = *new* data-type;
  //declaring a pointer;
  data-type* var-name;
  //later allocating memory
  var-name = *new* data-type;

To release/ free the memory :

*delete* var-name;

```
51
52      int *ptr = new int;
53
54      //freeing memory
55      delete ptr;
56
57      ptr = NULL;
58
```

```
52      int *ptr;
53      ptr = new int;
54
55      //freeing memory
56      delete ptr;
57
58      ptr = NULL;
59
```

- For array length that are not know before runtime, they can be allocated memory dynamically.
- Format :

  data-type* var-name = *new* data-type[Len];
- To release/ free the memory :

  *delete* [] var-name;

```
52      int *ptr = new int[LEN];
53
54      //freeing memory
55      delete [] ptr;
56
57      ptr = NULL;
58
```

# Passing Pointers to functions


- With passing an argument to a function we have learnt so far. We know that whenever we pass a value to a function a copy of that value is created and passed to that function
- Hence, the original value is left unchanged even after the function has executed.
- Here in main(), we have a variable val =2; [line 11] we call the increment(val) on it [line 15].
- In the increment() the value passed is incremented by 1 [line 6]
- After the code is executed the value of val remains unchanged. Why?
- Because a copy of val was made and passed to the increment() so any changes made to that value does not affect the original one.
- As we can see in the results below.

```
1   #include <iostream>
2
3   using namespace std;
4
5   void increment(int x) {
6       x++;
7   }
8
9   int main() {
10
11  int val = 2;
12
13  cout << "Before increment : " << val << endl;
14
15  increment(val); // the value of val is not changed
16
17  cout << "After increment : " << val << endl;
18
19      return 0;
20  }
```

```
PS C:\Users\ROOT\Documents\CPP community\pointers> g++ -o pointer pointer.cpp
PS C:\Users\ROOT\Documents\CPP community\pointers> ./pointer.exe
Before increment : 2
After increment : 2
PS C:\Users\ROOT\Documents\CPP community\pointers>
```

# Passing Pointers to functions


- When a function has  a pointer as a parameter, in passing an argument the address of the argument is passed to the function.
- A pointer takes the address of a variable.
- On lines 91 to 93 the increment() has be altered to take a pointer as an parameter.
- Notice [line 92] the value is first dereferenced before the increment is done.
- Here the location of the original value is passed to the function, hence any changes made to that value affects the original value too.
- As seen in the results below.

```
76  int x = 2;
77
78  increment_x (x);
79  cout << "value after normal increment x : " << x << endl << endl;;
80  ptr_increment_x(&x);
81  cout << "value after pointer increment x : " << x << endl;
82
83      return 0;
84  }
85  //x is passed by value ie. copy is passed to function
86  void increment_x (int x) {
87      //increments the value of x by 1
88      x++;
89  }
90  //x is passed by reference ie. address is passed to function
91  void ptr_increment_x (int* x) {
92      (*x)++; // increments the value of x by 1
93  }
```

Pass by value function

Pass by pointer

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE   1: powershell

PS C:\Users\ROOT\Documents\CPP community\pointers> g++ -o pointer pointer.cpp
PS C:\Users\ROOT\Documents\CPP community\pointers> ./pointer.exe
Before increment : 2
After increment : 3
PS C:\Users\ROOT\Documents\CPP community\pointers>
```


# Try Work 03

Implement the swap function using pointers as parameters.

- Line 9: a variable is created to hold value temporarily called temp.
- Line 10: the value of first is copied to the temp.
- Line 11: the value of second is copied to first, so now first has the same value as second.
- Line 12: To complete the swap the value of temp is copied to second.
- Now, first holds the initial value of second and second holds the initial value of first.

```
8   void swap (int first, int second) {
9       int temp;
10      temp = first;
11      first = second;
12      second = temp;
13
14  }
15
```


# References

- A reference is an alias to another variable, or an alternative name to another variable.
- Remember in high school where everybody had a nickname? Yeah! Nicknames are references to them.
- For instance my name is `Yaw Ofori` a.k.a `AwesomeKen`. Whenever you here `AwesomeKen` I pop up in your mind and you know `AwesomeKen` and `Yaw Ofori` is the same person. So any thing that happens to `AwesomeKen` has happened to `Yaw Ofori`.
- Wouldn't you find it ridiculous to find a nickname that does not refer to anyone? Surely, I would and you should! There certainly should be a body to bear that nickname.
- In the same vane, a reference must refer to an object, it cannot refer to a null object. Also, it cannot be declared and later defined, it must be initialized on creation.

Format :

```
data-type& ref-name = object-name;
```

# References

- References are much like pointers but not entirely. They are much easier to use when the concept is grasped well.
- It also targets the memory address of the variable and hence any changes made to the reference also affects the original variable.
- When references are used as parameters to a function, the compiler takes care of passing the address of the argument to the reference variable.

```
16  int main {
17
18      int age = 39;
19
20      int& refAge = age;
21
22
23      int& refSomething = NULL;
24
25      int old = 21;
26
27      int& refOld;
28
29      refOld = old; //This generates an error
30
```

Creating a reference

A reference cannot refer to a NULL object.

A referenced must be assigned on declaration. Once assigned it cannot be reassigned.

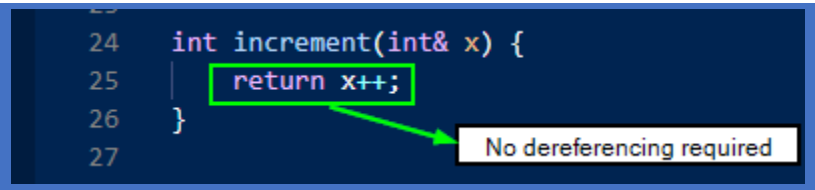
# Passing References to Functions

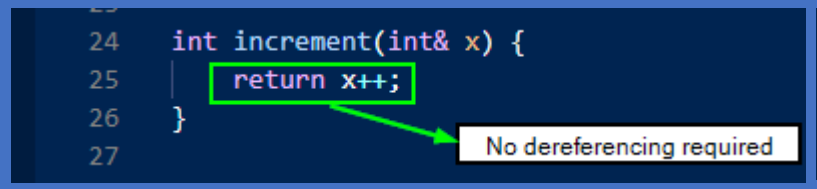
This is similar to passing pointers, however with references the address-of operator & is used as opposed to * operator for pointers.

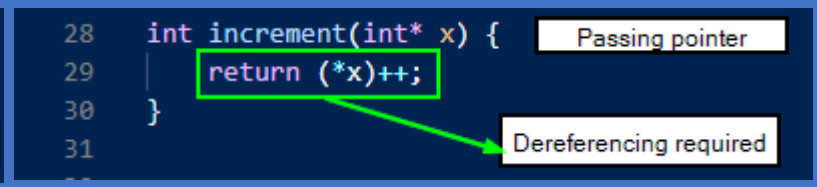
Format :

int doSomething(int& var1, float& var2);

Since, with references an alternative name is being created. We do not have to worry about dereferencing before we can perform operations as with pointers. We simply use the new aliases.



# Try Work 04

- Implement the swap function using references as parameters.
- Line 9: a variable is created to hold value temporarily called temp.
- Line 10: the value of first is copied to the temp.
- Line 11: the value of second is copied to first, so now first has the same value as second.
- Line 12: To complete the swap the value of temp is copied to second.
- Now, first holds the initial value of second and second holds the initial value of first.

```
 8   void swap (int first, int second) {
 9       int temp;
10       temp = first;
11       first = second;
12       second = temp;
13
14   }
15
```


# Questions

- char test = 127; test++;
  what is the value of test after increment?
- When a variable is declared const at what point can the value be modified in the code?
- Why are pointers dereferenced?
- Referenced arguments have their addresses automatically passed to functions? True or False.


# Summary

- Type modifiers allow you to alter the storage capacity of the various datatypes to accommodate the size of the data to be stored.
- All datatypes are by default signed ie the can hold both signed ( positive and negative ) values.
- Unsigned storage can only hold positive values
- Signed storage size is half the size of unsigned storage size.

# Summary


- There are two kinds of memory allocation in C/C++; Static and Dynamic Memory Allocation.
- Static is allocated at compile time and dynamic is allocated during runtime.
- Size of static memory must be known at compile time while dynamic memory's size is decided at runtime.
- Pointers are variables that hold memory address of other variables or memory locations. They allow for dynamic memory allocation.
- Memory dynamically allocated (using keyword new) must be freed (using keyword delete) to avoid memory leaks.


# Summary

- A reference is an alias to a variable.
- Unlike pointers, once a reference is assigned it cannot be re-assigned.
- Reference and pointers are good and efficient ways to pass huge data structures to functions to prevent copying the whole data structure to the function.

babystepz

# Thank **YOU**