SINAI UNIVERSITY

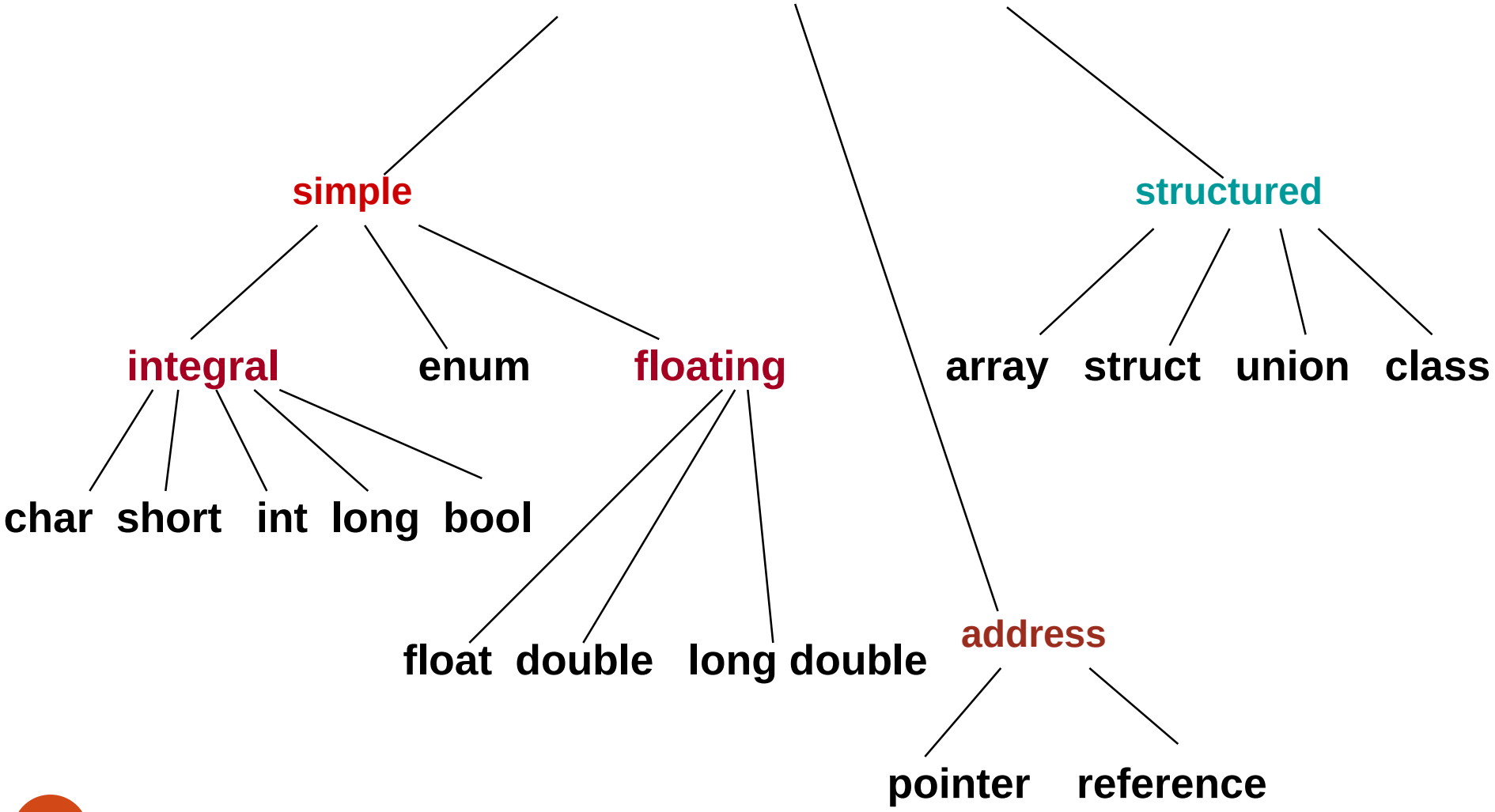
CSW 232
Computer Programming (1)

SPRING 2024

Lecture 09 – Pointers

Instructor: Dr. Tarek Abdul Hamid

C++ Data Types


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

Pointers

- Pointers are among C++'s most powerful, yet most difficult concepts to master.
- We've seen how we can use references to do Pass-By-Reference.
- We can use Pointers to create Dynamic Data structures like Linked Lists, Stacks, Queues and Trees.

Pointers

- Pointers are a type of variable, just like int, double, etc., except instead of storing a value, they store a **memory address** of another variable.

- In this sense, a variable **directly** references a value, and a pointer **indirectly** references a value.

Pointers

- Pointers, just like other variables, must be declared before they can be used. For example, the declaration

```
int *countPtr;
```

- Declares a variable countPtr to be of type int * (a **pointer** to an **int** value)
- This is read as "countPtr is a pointer to an int".
- Each variable being declared as a pointer must be preceded by an asterisk (*).
- Also, although not required, declaring any pointer value with the name ending in Ptr is a good idea.

Initializing Pointers

- A Pointer may be initialized to 0, NULL, or an address.
- NULL is a Symbolic constant defined in <iostream> to represent the value 0.
- A Pointer that is assigned 0 or NULL points to nothing.

Initializing Pointers

- The *address operator*, (**&**), is a unary operator that returns the memory address of it's operand.

- This is how we can assign a memory address to a pointer variable.

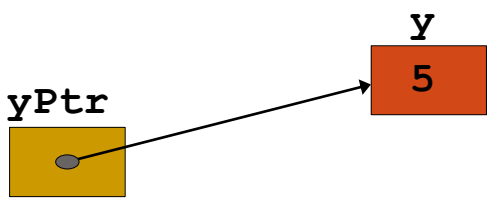
```
int y=5;          //Declare an int, assign the value of 5
int *yPtr;        //Declare a pointer to an int
yPtr = &y;        //Assign yPtr the memory address of y
```

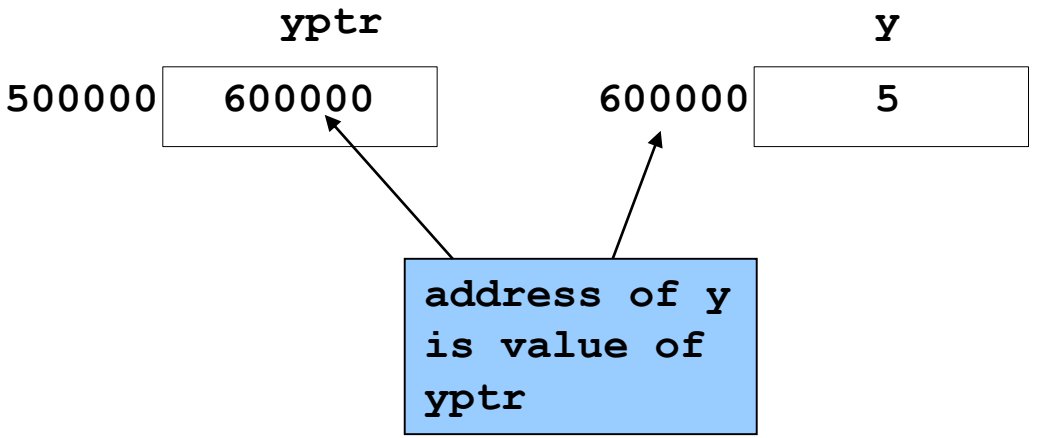
Pointer Operators

- Example

```
int y = 5;
int *yPtr;
yPtr = &y;     // yPtr gets address of y
```

- **yPtr** "points to" **y**





The Dereferencing Operator

- The * operator, referred to as the indirection, or dereferencing operator, returns an alias to what the pointer is pointing to.

- In the previous example, the line

  cout <<*yPtr; //Will Print 5

- Will print the value of the variable that *yPtr points (which is y, which is 5)

- Basically, *yPtr "returns" y

- Operations like below are also legal

  *yPtr = 7; //Changes y to 7

Address of and Dereference

- The address of (&) and dereference (*) operators are actually inverses of each other.

- They cancel each other out

```
*&myVar == myVar
```

and

```
&*yPtr == yPtr
```

```
1  // Fig. 5.4: fig05_04.cpp
2  // Using the & and * operators
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int a;        // a is an integer
11    int *aPtr;    // aPtr is a pointer to an integer
12
13    a = 7;
14    aPtr = &a;    // aPtr set to address of a
15
16    cout << "The address of a is " << &a
17         << "\nThe value of aPtr is " << aPtr;
18
19    cout << "\n\nThe value of a is " << a
20         << "\nThe value of *aPtr is " << *aPtr;
21
22    cout << "\n\nShowing that * and & are inverses of "
23         << "each other.\n&*aPtr = " << &*aPtr
24         << "\n*&aPtr = " << *&aPtr << endl;
25    return 0;
26 }
```

The address of **a** is the value of **aPtr**.

The ***** operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

Notice how ***** and **&** are inverses

```
The address of a is 006AFDF4
The value of aPtr is 006AFDF4
The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 006AFDF4
*&aPtr = 006AFDF4
```

Example

Write a C++ Program using Pointer to swap 2 numbers without using 3rd variable

Example

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int a=20,b=10;
6  int *p1, *p2;
7  p1=&a;
8  p2=&b;
9
10 cout<<"Before swap: *p1="<<a<<endl;
11 cout<<"Before swap: *p2="<<b<<endl;
12
13 *p1 = *p1 + *p2;
14 *p2 = *p1 - *p2;
15 *p1 = *p1 - *p2;
16
17 cout<<"Before swap: *p1="<<b<<endl;
18 cout<<"Before swap: *p2="<<a<<endl;
19    return 0;
20 }
```

Function Passing

- Let's go back old topic. Passing arguments to functions.
- They are two well known ways – passing by value, and passing by reference.
- We can do pass by reference two different ways. Passing by Reference using references, or passing by reference using **pointers.**

Function Passing

- So let's think about this for a second. We're going to pass **variables**, by reference, using **pointers.**
- Pointers hold …?
- Memory Addresses
- and what operator did we just see that will give us the memory address of a variable?
- The ampersand (&) operator
- This is how we call a function that uses call by reference using pointers – we have to send it the memory address.. so the function call looks like

myfunc(&my_var); //calls function myfunc with mem address of my_var

Function Passing

- So we call a function that uses Pointer arguments with the syntax of

  myfunction(&my_var);

- So now we send a memory address to our function. To actually do anything with it, we need to deference it, both in the function definition and in the function body.

```
myfunction( int *my_varPtr )
{
    *my_varPtr = *my_varPtr * *my_varPtr;
}
```

Const Pointers

- Const Pointers, just like any const variable, is unable to be changed once it is initialized.
- Const Pointers are Pointers that always point to the same memory location.
- These must be initialized when they are declared.
- Remember, when your declaring a pointer variable, you have to declare it a type (int, double, etc.) – C++ also makes the distinction between a regular int and a const int.

Const Pointers

- So basically this leads to some screwy syntax with const pointers.. here it is.
- Assume x is declared as

  int x = 5;
- Non Constant Pointer to Non Constant Data
  - int *myPtr = x;
- Non Constant Pointer to Constant Data
  - **`const int *myPtr = &x;`**
- Constant Pointer to Non Constant Data
  - **`int *const myPtr = &x;`**
- Constant Pointer to Constant Data
  - **`const int *const Ptr = &x;`**

```
1  // Fig. 5.13: fig05_13.cpp
2  // Attempting to modify a
3  // non-constant data
4  #include <iostream>
5
6  int main()
7  {
8     int x, y;
9
10    int * const ptr = &x; // ptr is a constant pointer to an
11                          // integer. An integer can be modified
12                          // through ptr, but ptr always points
13                          // to the same memory location.
14    *ptr = 7;
15    ptr = &y;
16
17    return 0;
18 }
```

Changing ***ptr** is allowed - **x** is not a constant.

Changing **ptr** is an error - **ptr** is a constant pointer.

```
Error E2024 Fig05_13.cpp 15: Cannot modify a const object in function
main()
```

Pointer Arithmetic

- Welcome to the world of weird programming errors.
- This is another example of the powerful but dangerous nature of pointers.
- Pointer Arithmetic is so error prone, it's not allowed in Java or C#.
- This doesn't mean don't use – you may have, or at least understand it so you can understand other people's code. So master this.

Pointer Arithmetic

- So again, what is an array really?
- That's right, a const pointer.
- So, we can create a Pointer to the first element of an array with code like

```
int b[5] = { 0 };
int *bPtr;
bPtr = b;
* note, above line equivalent to bPtr = &b[0];
```

Pointer Arithmetic

- Normally, when we would want to access the 4th element in our array, we'd use notation like

  b[3];

  but, we can also do

  *(bPtr + 3); //actually does address of bPrt + 3 * 4*

  this is called using **Pointer/Offset notation.**

  We can also access pointers using subscripts

  bPtr[3]; // same as above

  this is called, you guessed it, **Pointer/Subscript notation.**

Thoughts on Previous

- Although you can use Pointer/Subscript notation on Pointers, and you can use Pointer/Offset notations with arrays, **try not to unless you have a good reason.**

- No technical reason, it is just confusing for people reading your code.

Null Pointers

- When you begin a program, you may have some pointers declared, but are not yet set to any value, in this case you should set them to NULL.
- NULL is a special value that indicates a pointer is unused.
- For each pointer type, there is a special value -- the "null pointer" -- which is distinguishable from all other pointer values and which is not the address of any object.
- NULL is defined as 0 (zero) in C++.

Why Null Pointers?

SU
SINAI UNIVERSITY

- When we declare (but not initialize) *ANY* variable, what value does it contain?
- What can't we do to a variable if we have no idea what it contains?
- Null Pointers give us a way to compare and see if something is initialized.

Comparing Pointers

- To test if a Pointer is null, you can either by either

```
int *intPtr=NULL;

if (intPtr==NULL)
```

or

```
if (intPtr== '0 ')
```

These both are equivalent. I like the first convention (more readable), but either is acceptable.

Arrays of Pointers

- Normally, we're used to Arrays containing ints, doubles, etc
- We can also make arrays of Pointers.
- This is most commonly seen with Arrays of C-Style strings.

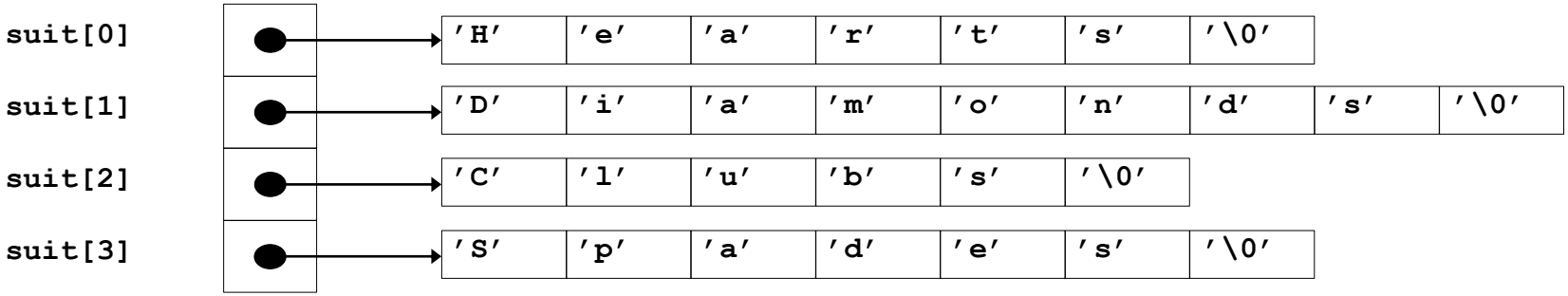
Array of Pointers

```
const char* suit =
  { "Clubs", "Diamonds", "Hearts", "Spades" }
```

- This basically says "each element is of type pointer to char"

Arrays of Pointers

- Arrays can contain pointers
  - Each element of not in the array, only pointers to the strings are in the array
  - suit is a pointer to a **`char *`** (a string)
  - The strings are



  - **`suit`** array has a fixed size, but strings can be of any size


Thanks!