CIS 129
Advanced Computer Programming

Chapter 6: Pointers

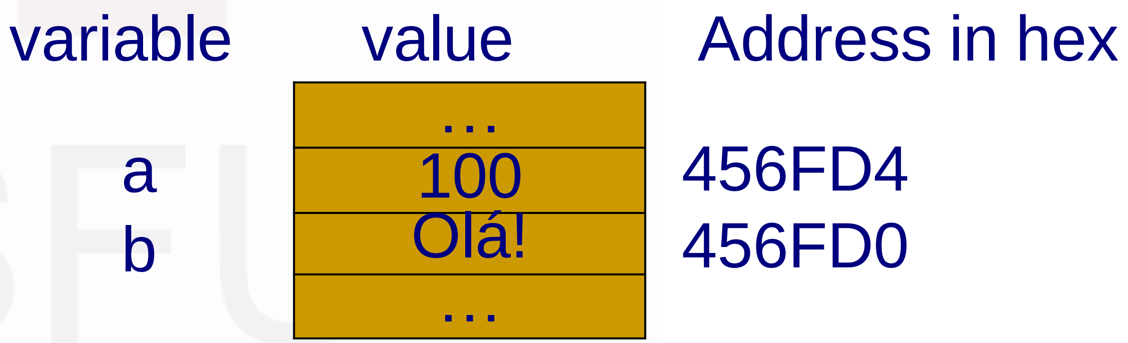
Mr. Horence Chan

SFU

Variables and Memory

- When you declare 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 _____________ that the variable name corresponds to
2. Go to that location in memory and retrieve or set the _____________ it contains

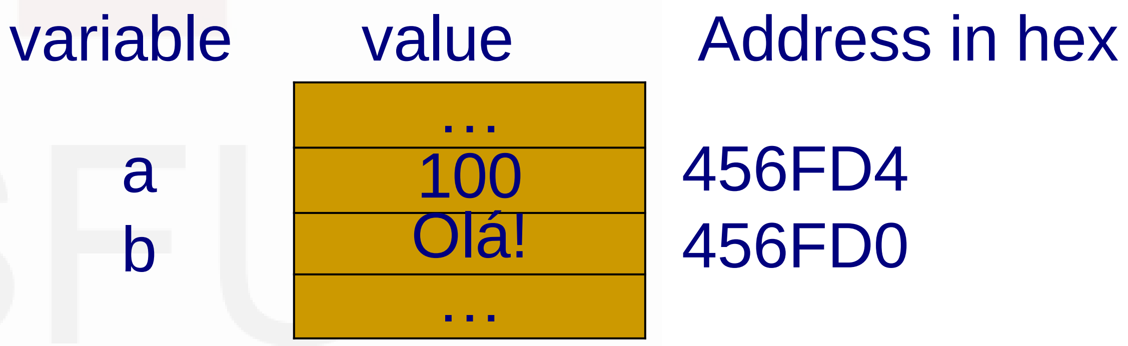


variable	value	Address in hex
	…	
a	100	456FD4
b	Olá!	456FD0
	…	

Variables and Memory

- C++ allows us to perform either one of these steps independently on a variable with the & and * operators:

1. __________ evaluates to the address of x in memory.
2. *(&x) takes the address of x and *dereferences* it – it retrieves the value at that ______________ in memory. *(&x) thus evaluates to the same thing as x.



variable	value	Address in hex
	…	
a	100	456FD4
b	Olá!	456FD0
	…	

Motivating Pointers

- Memory addresses, or *pointers*, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself. Just a taste of what we'll be able to do with pointers:
    - More flexible pass-by-reference
    - Manipulate complex data structures efficiently, even if their data is scattered in different memory locations
    - Use polymorphism – calling functions on data without knowing exactly what kind of data it is

Declaring Pointers

- To declare a pointer variable named ptr that points to an integer variable named `x`:
- `int *ptr = &x;`
- `int *ptr` declares the pointer to an integer value, which we are initializing to the address of `x`.
- We can have pointers to values of any type. The general scheme for declaring pointers is:
-
  ```
  data_type *pointer_name; // Add "= initial_value "
                           // if applicable
  ```
- `pointer_name` is then a variable of type data type `*` – a “pointer to a data type value.”

Using Pointer Values

- Once a pointer is declared, we can dereference it with the * operator to access its value:

-
  ```
  cout << *ptr; // Prints the value pointed to by ptr,
                // which in the above example would be
                //x’s value
  ```
- We can use deferenced pointers as values:

-
  ```
  *ptr = 5; // Sets the value of x
  ```
- Without the `*` operator, the identifier x refers to the pointer itself, not the value it points to:

-
  ```
  cout << ptr; // Outputs the memory address of x
               // in base 16
  ```

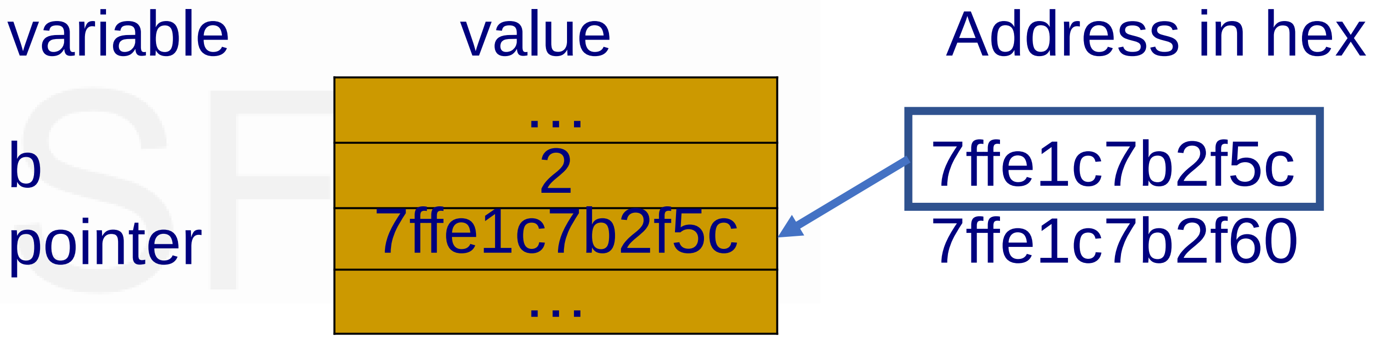
SFU

Using Pointer Values

```
#include <iostream>
using namespace std;
int main(){
int b = 2;
int *pointer = &b;
cout << "Value of b: " << b << endl;
cout << "Address of b: " << &b << endl;
cout << "Value of pointer：" << pointer << endl;
cout << "Address of pointer：" << &pointer << endl;
cout << "Value of *pointer：" << *pointer << endl;
return 0;}
```

Sample Output:

```
Value of b:
Address of b:
Value of pointer：
Address of pointer：
Value of *pointer：
```



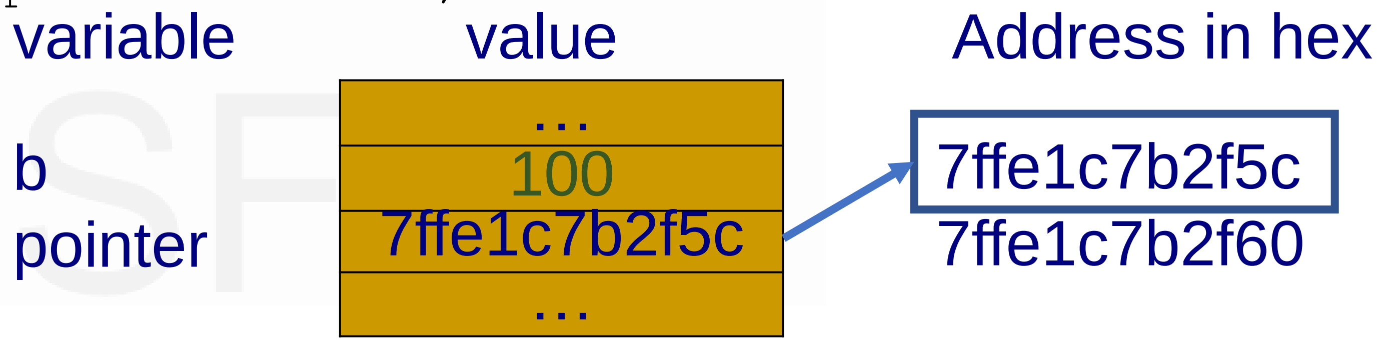
variable	value	Address in hex
	…	
b	2	7ffe1c7b2f5c
pointer	7ffe1c7b2f5c	7ffe1c7b2f60
	…	

Using Pointer Values

```
#include <iostream>
using namespace std;
int main(){
int b = 2;
int *pointer = &b;
*pointer = 100;
cout << "Value of b: " << b << endl;
cout << "Address of b: " << &b << endl;
cout << "Value of pointer：" << pointer << endl;
cout << "Address of pointer：" << &pointer << endl;
cout << "Value of *pointer：" << *pointer << endl;
return 0;}
```

Sample Output:

```
Value of b:
Address of b:
Value of pointer：
Address of pointer：
Value of *pointer：
```



Using Pointer Values

- 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){...}`.

…

```
void squareByPtr (int * numPtr) {
*numPtr = * numPtr  *  * numPtr ;
}
int main () {
int x =  5;
squareByPtr (&x);
cout << x; //     Prints 25
}
```

(* : Multiply operator)

Null and uninitialized pointers

- Pointer need to initialize by assigning it a valid __________, pointer cannot declared without initialization

```
int *ptr;
*ptr =55;
```

```
int a;
int *ptr =&a;
*ptr =55;
```

- Pointer can be initialize to _____ or NULL, pointer need to assign to a valid address afterwards, if not dereferencing that pointer will cause error.

```
int *ptr = 55;
int a;
ptr = &a;
```

```
int *ptr = 0;
int a;
ptr = &a;
*ptr =55;
```

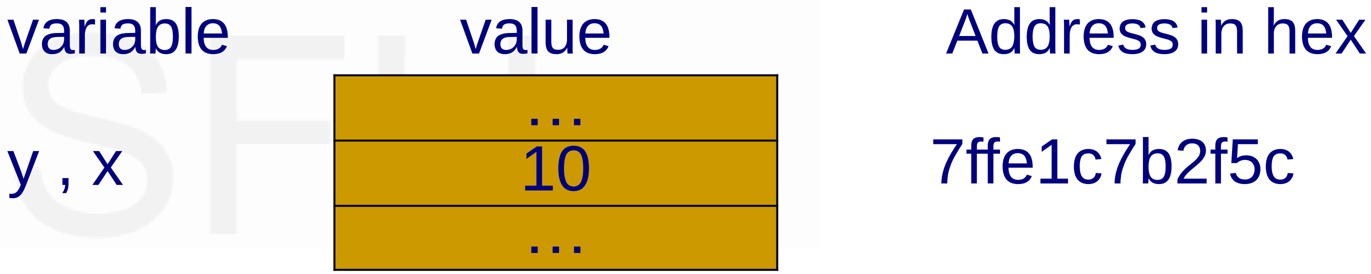
SFU

References

- 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 = 10;
int &x = y; // Makes x a reference to, or alias of, y
```

- After these declarations, changing x will change y and vice versa, because they are two names for the _________________.



variable	value	Address in hex
	…	
y , x	10	7ffe1c7b2f5c
	…	

References

- References are just pointers that are dereferenced every time they are used. Just like pointers, you can pass them around, return them, set other references to them, etc.
- The differences between using pointers and using references are:
  - When writing the value that you want to make a reference to, you do not put an _______ before it to take its address, whereas you do need to do this for pointers.

Reference		Pointer

`int y = 10;`
`int& x = &y;`
 	
	`int y = 10;`
	`int& x = y;`
	
		`int a;`
		`int *ptr =&a;`

References

- The differences between using pointers and using references are:
  - You ____________ change the location to which a reference points, whereas you ____________ change the location to which a pointer points. Because of this, references must always be initialized when they are declared.

Reference	Pointer

`int y = 10;`
`int z = 20;`
`int& x = y;`
`& x = z;`
 	
	`int y = 10`
	`int z = 20;`
	`int * x = &y;`
	`x = &z;`

* operator

1. When ______________ a pointer, `*` is placed before the variable name to indicate that the variable being declared is a pointer – say, a pointer to an `int` or `char`, not an `int` or `char` value.

   (e.g. `int * pointer = &b;`)

2. When using a pointer that has been set to point to some value, * is placed before the pointer name to ______________ it – to access or set the value it points to.

   (e.g. `*pointer = 100;`
   `cout<< *pointer;`)

& operator

1. To indicate a ___________ data type

   (e.g. `int &x = y;`)

2. To take the ____________ of a variable

   (e.g. `int *ptr = &x;`)

SFU

Pointers and Arrays

```
long arr[] = {6, 0, 9, 5};

long *ptr = arr; //Point to ________ element of
                        array
cout << "arr[0] =  " << *ptr<< endl;
```

- The name of an array is actually a pointer to the ___________ element in the array.

Output:

arr[0] = ______

```
ptr++;
cout << "arr[1] = " << *ptr<< endl;
```

```
long *ptr2 = arr + 3; // Point to _____ element
                                of array
cout << "arr[3] = " << *ptr2<< endl;
```

- Writing `myArray[3]` tells the compiler to return the element that is 3 away from the starting element of `myArray`.

Output:

arr[3] = ______

```
cout<< "No. of array element between ptr2 and
ptr: "<<(ptr2-ptr);
```

Pointer Step Size

```
long arr[] = {6, 0, 9, 5};

long *ptr = arr; //Point to ________ element of
                  array

cout << "arr[0] =  " << *ptr<< endl;

ptr++; //Point to ________ element of array

cout << "arr[1] = " << *ptr<< endl;

long *ptr2 = arr + 3; // Point to ________
                        element of array

cout << "arr[3] = " << *ptr2<< endl;

cout<< "No. of array element between ptr2 and
ptr: "<<(ptr2-ptr);
```

Complete Output:

arr[0] = ______

arr[1] = ______

arr[3] = ______

No. of array element between ptr2 and ptr:

______

SFU

Array Access Notations

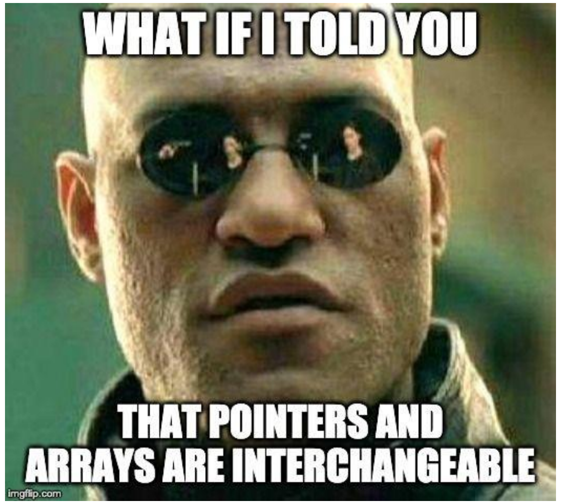
```
long arr[] = {6, 0, 9, 5};

long *ptr = arr;

cout << "arr[0] =  " << *ptr<< endl;

ptr++; cout << "arr[1] = " << *ptr<< endl;

long *ptr2 = arr + 3;

cout << "arr[3] = " << *ptr2<< endl;
cout << "arr[3] = " << arr[3]<< endl;
cout<< "No. of array element between ptr2 and
ptr: "<<(ptr2-ptr);
```

- *Array-subscript notation* (the form `arr3[3]`)can be used with pointers as well as arrays.
- When used with pointers, it is referred to as *pointer-subscript notation*.
- For instance, an alternate and functionally identical way to express `arr3[3]` is ______________________

- Output

arr[3] = ______

arr[3] = ______



char * Strings

```
char arr[] = { 'A', 'n', ' ', 'Y', 'e', 'o' , 'n' , 'g'};
char* ptr = arr + 3;
*ptr = 'D';
ptr++;
*ptr = 'w';
ptr++;
*ptr = 'a';
ptr++;
*ptr = 'e';
ptr++;
*ptr = '!';
ptr = arr;
for (int i = 0; i < 8; i++) {
        cout << *ptr;
        ptr++;}
```

- For simplicity, we can also write `arr[]` = _________________; in the beginning

- We can modify the contents of an array of characters.
- Attempting to modify one of the elements each time in `arr[]` is permitted

Output:

____

Array size

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

int main() {

int arr[] = {10, 20, 30, 40, 50};
int arrSize = *(&arr + 1) - arr;
cout << "The length of the array is: "
        << arrSize;

 return 0;
}
```

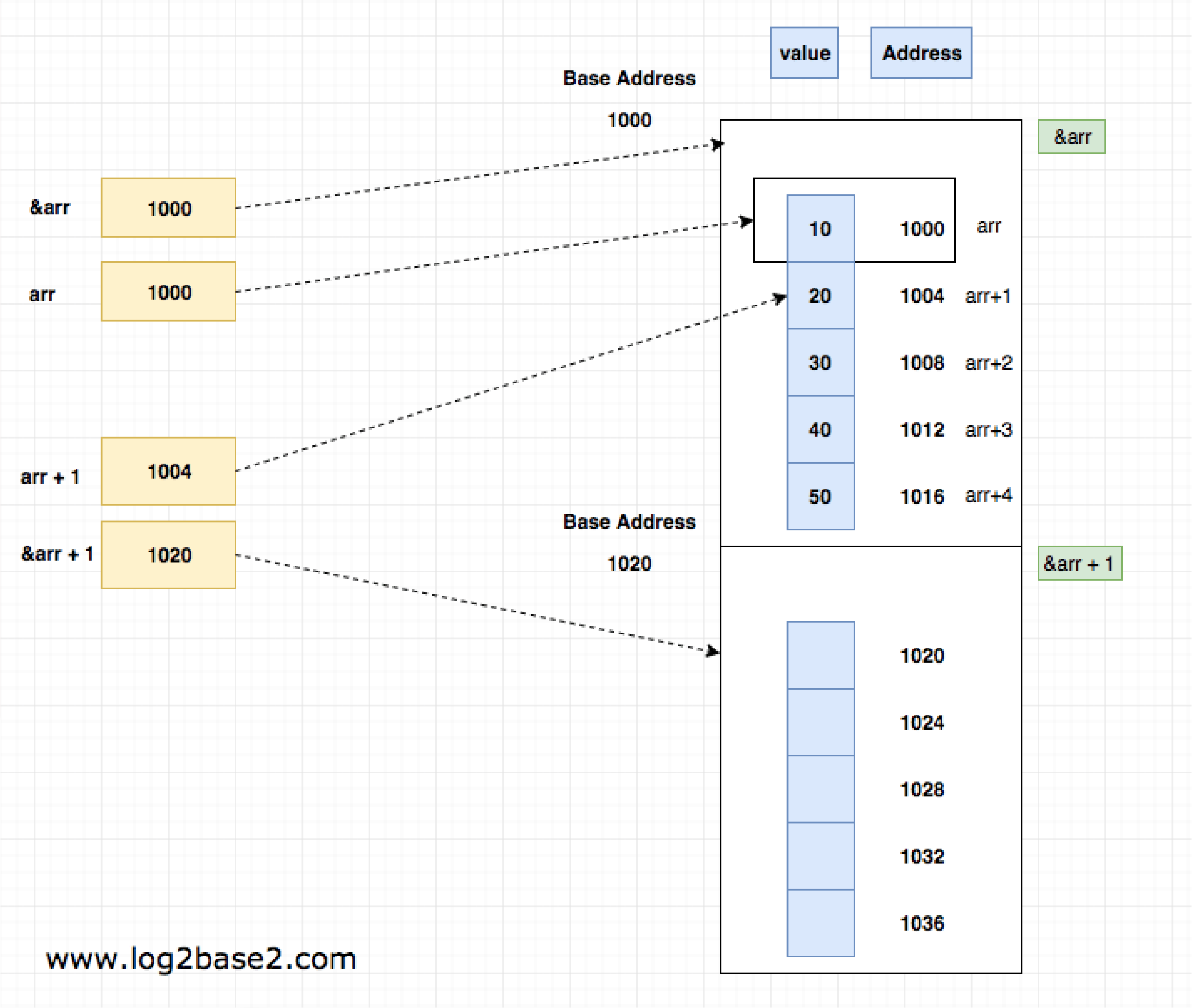
- Since we have a pointer at the start of the array
- The __________ of the array can be calculated if we manage to find out the address where the array __________.

- `&arr` is a pointer to an ____________ array, if we move `&arr` by 1 position it will point the next block of 5 elements `(&arr + 1)`
- `*(&arr + 1)` simply casts the above address to an `int *`.
- Subtracting the address of the __________ of the array, from the address of the __________ of the array, gives the __________ of the array.

- Output
- The length of the array is: ______

Array size

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

int main() {

int arr[] = {10, 20, 30, 40, 50};
int arrSize = *(&arr + 1) - arr;
cout << "The length of the array is: "
        << arrSize;

 return 0;
}
```



Dynamic Array

- Consider a regular array in C++,

```
int x[5]
```

- Once an array has been created, its _________ cannot be changed.
- It is allocated a predetermined amount of memory

- Dynamic array is different, its size is ________________ during program runtime. Dynamic array elements occupy a contiguous block of memory
- Dynamic array grows its memory size by a certain factor when there is a need

SFU

`new` and `delete`

- __________ a dynamic array using the `new` keyword.
- `pointer_variable = new data_type;`

- E.g. `int *arr = new int[n];`
- (n: size of array)

Me: I forgot to free memory..
you will take care of it?
C++ :
No, I don't think I will.

- Dynamic array should be __________ from the computer memory once its purpose is fulfilled
- The released memory space can then be used to hold another set of data
- `delete [] arr;`

SFU

Dynamic Array

```
#include<iostream>
using namespace std;
int main() {
        int x, n;
        cout << "How many numbers will you type?" << "\n";
        cin >> n;
        ____________________________
        cout << "Enter " << n << " numbers" << endl;
        for (x = 0; x < n; x++) {
                cin >> arr[x];
        }
        cout << "You typed: ";
        for (x = 0; x < n; x++) {
                cout << " " << arr[x];
        }
        cout << endl;
        ___________________________
        return 0;
}
```

- Create a dynamic array according to the size input by the user

- Delete dynamic array from the computer memory

SFU