Lecture 15

Pointers

Shibo Li

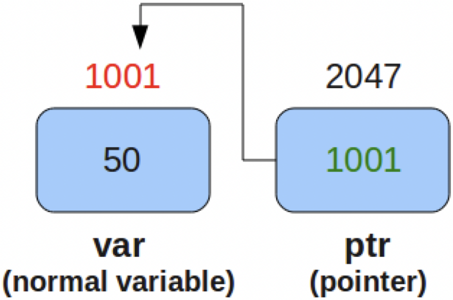
shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

What is a Pointer?

- A pointer is a variable that stores a memory address.
- Pointers are used to store the addresses of other variables or memory items.



Why we need Pointers?

FSU

- Pointers allow direct access and manipulation of memory. This is crucial in performance-critical application.
- Pointers are used to manage dynamic memory. This allows for the creation of variables or objects at runtime, which is essential when the size of data structures (like arrays) isn't known at compile time.
- When passing large objects (like structs or classes) to functions.

Declaring pointers:

- Pointer declarations use the * operator. They follow this format:

```
typeName * variableName;
int n; // declaration of a variable n
int * p; // declaration of a pointer, called p
```

- In the example above, p is a pointer, and its type will be specifically be referred to as "pointer to int", because it stores the address of an integer variable. We also can say its type is: `int*`

Declaring pointers:

- The **type** is important. While pointers are all the same size, as they just store a memory address, we have to know what kind of thing they are pointing TO.

```
double * dptr; // a pointer to a double
char * c1; // a pointer to a character
float * fptr; // a pointer to a float
```

Declaring pointers:

- Sometimes the notation is confusing, because different textbooks place the * differently. The three following declarations are equivalent:

```
int *p;
int* p;
int * p;
```

All three of these declare the variable p as a pointer to an int

- Another tricky aspect of notation: Recall that we can declare mulitple variables on one line under the same type, like this:

```
int x, y, z; // three variables of type int
```

- Since the type of a "pointer-to-int" is (int *), we might ask, does this create three pointers?

```
int* p, q, r; // what did we just create?
```

- NO! This is not three pointers. Instead, this is one pointer and two integers. If you want to create mulitple pointers on one declaration, you must repeat the * operator each time

Pointer dereferencing

- Once a pointer is declared, you can refer to the thing it points to, known as the target of the pointer, by ”dereferencing the pointer”. To do this, use the unary * operator:

```
int * ptr; // ptr is now a pointer-to-int
// ptr refers to the pointer itself
// *ptr the dereferenced pointer -- refers now to the
    TARGET
```

Pointer dereferencing

- Suppose that ptr is the above pointer. Suppose it stores the address 1234. Also suppose that the integer stored at address 1234 has the value 99.

```
cout << "The pointer is: " << ptr;
cout << "The target is: " << *ptr;
```

Pointer dereferencing

- Suppose that ptr is the above pointer. Suppose it stores the address 1234. Also suppose that the integer stored at address 1234 has the value 99.

```
cout << "The pointer is: " << ptr; // prints
    the pointer 1234
cout << "The target is: " << *ptr; // prints
    the target 99
```

- Note: the exact print out of an addres may vary based on the system.

Pointer dereferencing

- The notation can be a little confusing.
- If you see the * in a declaration statement, with a type in front of the *, a pointer is being declared for the first time.
- AFTER that, when you see the * on the pointer name, you are dereferencing the pointer to get to the target.

Pointer dereferencing

- Pointers don't always have valid targets.
  - A pointer refers to some address in the program's memory space.
  - A program's memory space is divided up into segements
  - Each memory segment has a different purpose. Some segments are for data storage, but some segments are for other things, and off limits for data storage

Pointer dereferencing

- Pointers don't always have valid targets.
  - If a pointer is pointing into an "out-of-bounds" memorysegment, then it does NOT have a valid target (for your usage)
  - If you try to dereference a pointer that doesn't have a valid target, your program will crash with a **segmentation fault** error. This means you tried to go into an off-limits segment

Initializing Pointers

So, how do we initialize a pointer? i.e. what can we assign into it?

```
int * ptr;
ptr = ; // with what can we fill this blank?
```

- The null pointer
- Pointers of the same type
- The “address of” operator
- Reinterpreted pointer of a different type
- Address to a dynamically allocated chunk of memory.

The null pointer

- here is a special pointer whose value is 0. It is called the null pointer
- You can assign 0 into a pointer:

```
int * ptr;
ptr = 0;
```

- The null pointer is the only integer literal that may be assigned to a pointer. You **may NOT assign arbitrary numbers to pointers**

The null pointer

- You **may NOT assign arbitrary numbers to pointers**

```
int * p = 0; // OK assignment of null pointer to p
    int * q;
q = 0; // okay. null pointer again.
int * z;
z = 900; // BAD! cannot assign other literals to
    pointers!
double * dp;
dp = 1000; // BAD!
```

The null pointer

- The null pointer is never a valid target, however. If you try to dereference the null pointer, you WILL get a segmentation fault.
- So why use it?

- The null pointer is typically used as a placeholder to initialize pointers until you are ready to use them (i.e. with valid targets), so that their values are known.
  - If a pointer's value was completely unknown – random memory garbage – you'd never know if it was safe to dereference
  - Make sure your pointer is ALWAYS set to either a valid arget, or to the null pointer, then you can test for it:

```
if (ptr != 0) // safe to dereference
   cout << *ptr;
```

Assigning Pointers of the same type

- It is also legal to assign one pointer to another, provided that they are the same type:

```
int * ptr1, * ptr2; // two pointers of type
int ptr1 = ptr2; // can assign one to the other
// now they both point to the same place
```

Assigning Pointers of the same type

- Although all pointers are addresses (and therefore represented similarly in data storage), we want the type of the pointer to indicate what is being pointed to. Therefore, C treats pointers to different types AS different types themselves.

```
int * ip; // pointer to int
char * cp; // pointer to char
double * dp; // poitner to double
```

- These three pointer variables (ip, dp, cp) are all considered to have different types, so assignment between any of them is illegal. The automatic type coercions that work on regular numerical data types do not apply:

```
ip = dp; // ILLEGAL
dp = cp; // ILLEGAL
ip = cp; // ILLEGAL
```

- As with other data types, you can always force a conversion by performing an explicit cast operation. With pointers, you would usually use reinterpret cast. Be careful that you really intend this, however!

```
ip = reinterpret_cast<int* >(dp);
```

The "address of" operator

- Recall, the & unary operator, applied to a variable, gives its address:

```
int x;
// the notation &x means "address of x"
```

- This is the best way to attach a pointer to an existing variable:

```
int * ptr; // a pointer
int num; // an integer
ptr = &num; // assign the address of num to ptr //
    now ptr points to "num"!
```

Pass by Address/Pointers

- We've seen that regular function parameters are pass-by-value
  - A formal parameter of a function is a local variable that will contain a copy of the argument value passed in
  - Changes made to the local parameter variable do not affect the original argument passed in

Pass by Address/Pointers

- If a pointer type is used as a function parameter type, then an actual address is being sent into the function instead
  - In this case, you are not sending the function a data value –instead, you are telling the function where to find a specific piece of data
  - Such a parameter would contain a copy of the address sent inby the caller, but not a copy of the target data
  - When addresses (pointers) are passed into functions, the function could affect actual variables existing in the scope of the caller

Example 1

```
void addone(int a){
   a = a + 1;
   cout << &a << endl; // ???
}

int main() {
   int x = 1;
   int* pt = &x; // pt = 1234
   addone(x);
   cout << x << endl; // ???
   cout << pt << endl; // ???
}
```

Example 1

```
void addone(int a){
   a = a + 1;
   cout << &a << endl; // the address of a (a copy
       of x)
}

int main() {
   int x = 1;
   int* pt = &x; // pt = 1234
   addone(x);
   cout << x << endl; // 1
   cout << pt << endl; // 1234
}
```

Example 2

```
void addone(int& a) {
   a = a + 1;
   cout << &a << endl; // ???
}

int main() {
   int x = 1;
   int* pt = &x; // pt = 1234
   addone(x);
   cout << x << endl; // ???
   cout << pt << endl; // ???
}
```

Example 2

```
void addone(int& a) {
   a = a + 1;
   cout << &a << endl; // 1234
}

int main() {
   int x = 1;
   int* pt = &x; // pt = 1234
   addone(x);
   cout << x << endl; // 2
   cout << pt << endl; // 1234
}
```

Example 3

```
void addone(int* a) {
   *a = *a + 1;
   cout << a << endl; // ???
   cout << &a << endl; // ???
}

int main() {
   int x = 1;
   int* pt = &x; // pt = 1234
   addone(x);
   cout << x << endl; // ???
   cout << pt << endl; // ???
}
```

Example 3

```
void addone(int* a) {
   *a = *a + 1;
   cout << a << endl; // 1234
   cout << &a << endl; // address of the pointer to a
}

int main() {
   int x = 1;
   int* pt = &x; // pt = 1234
   addone(x);
   cout << x << endl; // 2
   cout << pt << endl; // 1234
}
```

Memory Representation

- **Reference variable:** A reference does not actually have a distinct memory location from the variable it refers to. It's more of a compiler-level abstraction or alias. When a reference is created, it is essentially an alternate name for the memory location of the original variable. Thus, no additional memory is allocated to store a reference.
- **Pointer variable:** A pointer, on the other hand, is a separate entity in memory that holds the address of another variable. This means it occupies its own space, typically 4 or 8 bytes (depending on the system architecture, 32-bit or 64-bit).

Address Handling and Dereferencing

- **Reference variable:** At a low level, a reference is not directly manipulated using the address of the variable. Therefore, when the CPU accesses the value of a reference, it is the same as accessing the original variable. There's no need for an explicit dereferencing operation.
- **Pointer variable:** A pointer stores the memory address of another variable. The CPU needs to perform a dereferencing operation (*ptr) to access the value stored at the address. This involves fetching the address stored in the pointer, then performing a memory lookup to get the actual value at that address..

CPU Instructions and Operations

- **Reference variable:** Since references are resolved at compile-time, there is no additional overhead during runtime for dereferencing. The compiler directly generates code that works with the underlying variable, meaning the operations involving references are as efficient as using the original variable itself.
- **Pointer variable:** The use of pointers involves additional CPU instructions to load and dereference addresses. The CPU has to first load the memory address stored in the pointer, then access or modify the value at that address. This introduces an extra level of indirection and potentially more memory accesses.

Nullability and Safety

- **Reference variable:** At the assembly level, there is no null reference concept. A reference always refers to a valid object. Thus, the compiler assumes that references are never null, making references safer from memory corruption or runtime errors like dereferencing a null pointer.
- **Pointer variable:** Pointers, however, can be null (i.e., store a memory address of 0x0), and dereferencing a null pointer leads to undefined behavior, often causing a crash. The pointer's value must be checked at runtime to avoid null dereferences, adding complexity and potential runtime overhead.

Low-Level Flexibility

- **Reference variable:** References provide a higher-level abstraction and hide the complexities of memory manipulation. However, because of this, they lack the flexibility of pointers at a low level, especially in tasks involving memory management or dynamic allocation.
- **Pointer variable:** Pointers are more flexible at a low level because you can manipulate the memory address directly, increment or decrement pointer values (e.g., in array manipulation), and allocate or free memory manually (e.g., new and delete in C++).

- With a regular array declaration, you get a pointer for free. The name of the array acts as a pointer to the first element of the array.

```
int list[10]; // the variable list is a pointer//
    to the first integer in the array
int * p; // p is a pointer. same type as list.
p = list; // legal assignment. Both pointers to
    ints.
```

Pointers and Arrays:

- In the above code, the address stored in list has been assigned to p. Now both pointers point to the first element of the array. Now, we could actually use p as the name of the array!

```
list[3] = 10;
p[4] = 5;
cout << list[6];
cout << p[6];
```

Pointer Arithmetic

- Another useful feature of pointers is pointer arithmetic.
- When you add to a pointer, you do not add the literalnumber. You add that number of units, where a unit is the type being pointed to.
- Suppose ptr is a pointer to an integer, and ptr stores theaddress 1000. Then the expression (ptr + 5) does not give 1005 (1000+5).
- Instead, the pointer is moved 5 integers (ptr + (5 * size-of-an-int)). So, if we have 4-byte integers, (ptr+5) is 1020 (1000 + 5*4).

Another Example

```
void addone(int* a) {
   a = a + 1;
   cout << a << endl; // ???
   cout << &a << endl; // ???
   cout << *a << endl; // ???
}

int main() {
   int x = 1;
   int* pt = &x; // pt = 1234
   addone(x);
   cout << x << endl; // ???
   cout << pt << endl; // ???
}
```

Another Example

```
void addone(int* a) {
   a = a + 1;
   cout << a << endl; // 1238
   cout << &a << endl; // address of a
   cout << *a << endl; // dangerous
}

int main() {
   int x = 1;
   int* pt = &x; // pt = 1234
   addone(x);
   cout << x << endl; // 1
   cout << pt << endl; // 1234
}
```

Pointer Arithmetic

- In the above array example, we referred to an array item with p[6]. We could also say *(p+6).
- For instance, p + 6 in the above example means to move the pointer forward 6 integer addresses. Then we can dereference it to get the data *(p + 6).
- Most often, pointer arithmetic is used with arrays.

Pointer Arithmetic

What pointer arithmetic operations are allowed?

- A pointer can be incremented (++) or decremented (–)
- An integer may be added to a pointer (+ or +=)
- An integer may be subtracted from a pointer (- or -=)
- One pointer may be subtracted from another

- The fact that an array's name is a pointer allows easy passing of arrays in and out of functions. When we pass the array in by its name, we are passing the address of the first array element. So, the expected parameter is a pointer. Example:

```
// This function receives two integer pointers, //
    which can be names of integer arrays.
int Example1(int * p, int * q);
```

Pass By Address with arrays:

- When an array is passed into a function (by its name), any changes made to the array elements do affect the original array, since only the array address is copied (not the array elements themselves).

```
void Swap(int * list, int a, int b)
{
   int temp = list[a];
   list[a] = list[b];
   list[b] = temp;
}
```

- This Swap function allows an array to be passed in by its name only. The pointer is copied but not the entire array. So, when we swap the array elements, the changes are done on the original array. Here is an example of the call from outside the function:

```
int numList[5] = 2, 4, 6, 8, 10;
Swap(numList, 1, 4); // swaps items 1 and 4
```

- Note that the Swap function prototype could also be written like this:

```
void Swap(int list[], int a, int b);
```

Pass By Address with arrays:

- The array notation in the prototype does not change anything. An array passed into a function is always passed by address, since the array's name IS a variable that stores its address (i.e. a pointer).

Pass By Address with arrays:

Pass-by-address can be done in returns as well – we can return the address of an array.

```
int * ChooseList(int * list1, int * list2)
{ // returns a copy of the address of the array
   if (list1[0] < list2[0])
      return list1;
   else
      return list2;
}
```

And an example usage of this function:

```
int list1[5] = {1,2,3,4,5};
int list2[3] = {3,5,7};
int * p;
p = ChooseList(numbers, numList);
```

Using const with pass-by-address

- The keyword const can be used on pointer parameters, like wedo with references.
- It is used for a similar situation – it allows parameter passing without copying anything but an address, but protects against changing the data (for functions that should not change the original)
- The format:

```
const typeName * v
```

Using const with pass-by-address

- This establishes v as a pointer to an object that cannot be changed through the pointer v.
- Note: This does not make v a constant! The pointer v can bechanged. But, the target of v cannot be changed (through the pointer v).
- Example:

```
int Function1(const int * list); // the target
    of //list can't be changed in the function
```

Using const with pass-by-address

The pointer can be made constant, too. Here are the different combinations:

1. Non-constant pointer to non-constant data

```
int * ptr;
```

2. Non-constant pointer to constant data

```
const int * ptr;
```

Using const with pass-by-address

The pointer can be made constant, too. Here are the different combinations:

1. Constant pointer to non-constant data

```
int x = 5;
int * const ptr = &x; // must be initialized
    here
```

An array name is this type of pointer - a constant pointer (to non-constant data).

2. Constant pointer to constant data

```
int x = 5;
const int * const ptr = & x;
```

Using const with pass-by-address

The pointer can be made constant, too. Here are the different combinations:

1. Constant pointer to non-constant data

```
int x = 5;
int * const ptr = &x; // must be initialized
    here
```

An array name is this type of pointer - a constant pointer (to non-constant data).

2. Constant pointer to constant data

```
int x = 5;
const int * const ptr = & x;
```

const pointers and C-style strings

- We've seen how to declare a character array and initialize with a string:

```
char name[25] = "Spongebob Squarepants";
```

- Note that this declaration creates an array called name (ofsize 25), which can be modified.
- Another way to create a varible name for a string is to use just a pointer:

```
char* greeting = "Hello";
```

const pointers and C-style strings

- However, this does NOT create an array in memory that can be modified. Instead, this attaches a pointer to a fixed string, which is typically stored in a "read only" segment of memory (cannot be changed).
- So it's best to use const on this form of declaration:

```
const char* greeting = "Hello"; // better
```

Exercise: Subarray Sum

Goal: Write a program to find a continuous/consecutive subarray in an **unsorted** array that adds up to a given sum S.

```
// Given: {23, 17, 11, 2, 29, 40, 41, 39, 26, 10,
    42, 43};
```

- Return the starting and ending indexes of a subarray whose sum is equal to S
- If not found, print out not found such subarrays

```
Enter the target sum: 31
Subarray found between indexes 3 and 4
```

```
Enter the target sum: 71
Subarray found between indexes 3 and 5
```