CPSC 131
Data Structures

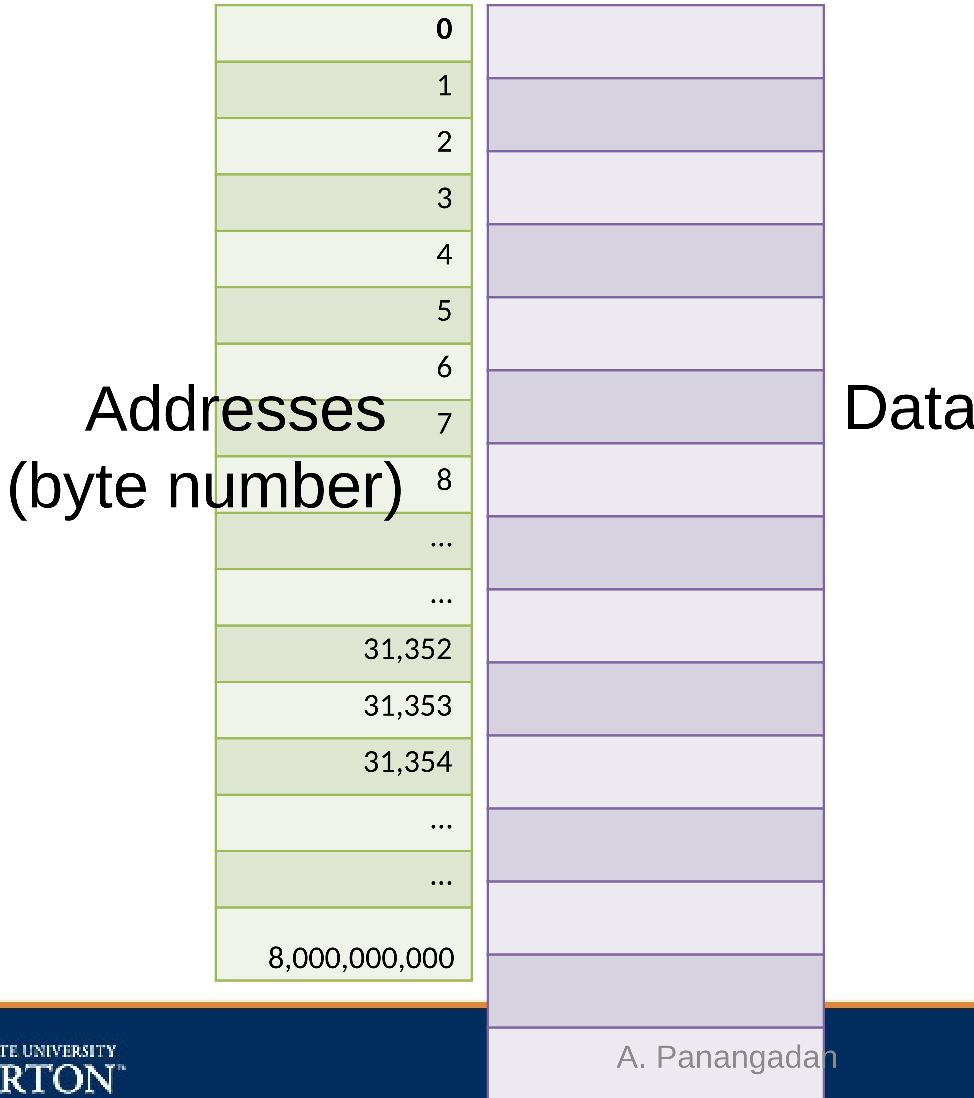
Dr. Shilpa Lakhanpal
shlakhanpal@fullerton.edu

Our topics for today

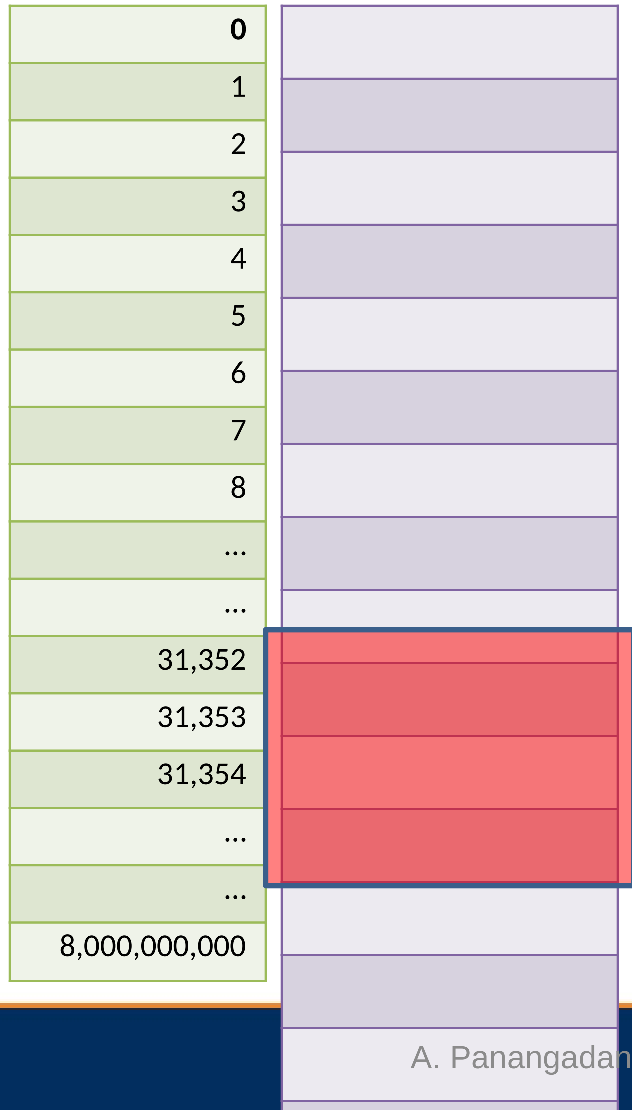
Review

- Vectors (or Arrays)
- Pointers
- References
- Dynamic memory

Data and Memory



Data and Memory




```
int n = 5;
```

Vector (or Array)

A vector (or array) stores a list of items in contiguous memory locations.

Advantage: Immediate / Random access to any element of vector (or array) v by using v.at(i) (or v[i])

Disadvantages:

- Slow Insert: Shifting elements to make room for new
- Slow Erase/Delete: Shifting elements to fill the gap left by deleted element

Linked List

A linked list is a list wherein each item contains not just data but also a **pointer**—a *link*—to the next item in the list.

Advantages: Fast inserts or deletes

Disadvantages:

- Access to i'th element may be slow

  (Why ?)

- Uses more memory due to storing a link for each item.

Pointer

There is:

A variable

And then there is:

Location of the variable

Pointers are variables that store

memory addresses of other variables

Usually a pointer will hold addresses of variables for a specific data type:

Example: Addresses of integer variables only, not for variables of type double.

Pointer syntax

Pointer variable declaration

DataType* PointerVariableName;

```
int*  intPointer;
```

means

`intPointer` is a variable and holds addresses of variables of integer data type.

Picture of a pointer variable

As with other variables, a pointer variable is represented by a box labeled by the variable name.

```
int* intPointer;
```

intPointer

&: Address-of Operator or <u>Reference operator</u>

The address-of operator returns the address of the variable that follows it.

Ex. If `number` is an integer variable,

`&number` will return the actual memory address of the variable `number`.

So

```
intPointer = &number;
```

means that `intPointer` will hold the address of `number`.

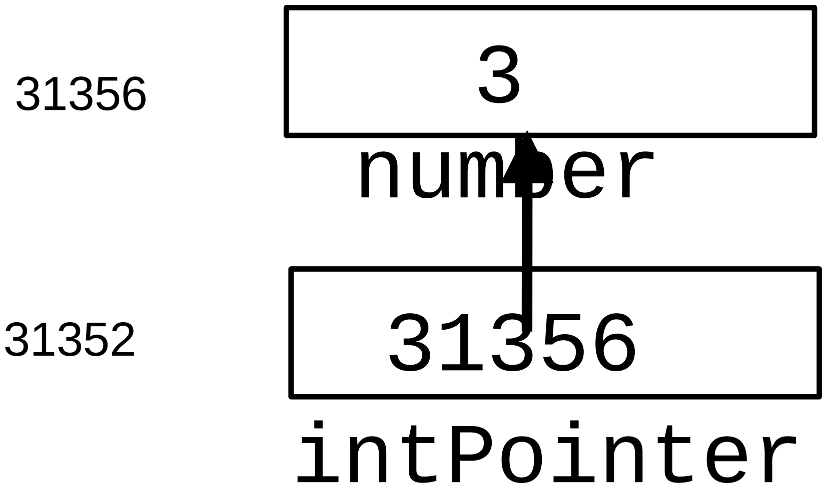
Read, `“intPointer is equal to the address of number”`

Picture of previous code

```
int number = 3;
int* intPointer;
intPointer = &number;
```

Assume memory location at address 31356 is reserved for number

Assume memory location at address 31352 is reserved for `intPointer`

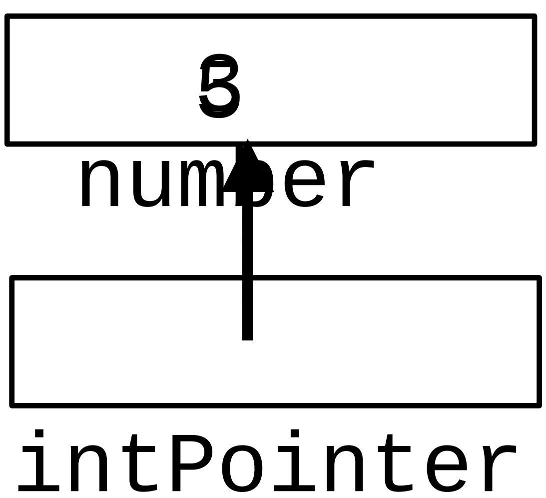


Since we don’t know the actual addresses, we usually depict the value stored in a pointer with an arrow pointing to the variable whose address it stores.

Can also say “`intPointer` **points** to the variable `number`.”

*: Dereference Operator

***** used with a pointer variable is an operator that returns the variable the pointer points to, so in the previous picture:



`*intPointer` is the same as `number` and

`*intPointer = 5;`
will mean

Points to think about

- What is the difference between:
  - Int* a, b, c
  - Int * a, b, c
  - Int *a, b, c

Points to think about

- In terms of language, nothing:
  - Int* a, b, c; Int * a, b, c; Int *a, b, c
    are all same and also same as
    Int b, c, *a,
    where a is a pointer to an int, and b
    and c are simple ints.

Points to think about

- Matter of preference:
  - To use int* a or int *a

Confusion only, when declaring multiple variables in one line

Dynamic Memory

We can create new objects while the program is running by using pointers.

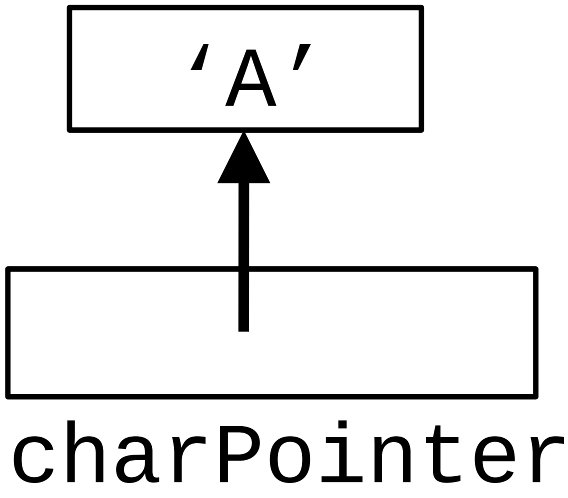
```
intPointer = new int;
```

We need a pointer variable to hold the address of the newly created dynamic object because we don't know where in memory the new object will be until the program is running!

Also called Heap memory or Free store

Picture of creating object in dynamic memory

```
char* charPointer;
charPointer = new char;
*charPointer = ‘A’;
```



Removing a dynamically allocated object

Static variables disappear when the function it is inside ends.

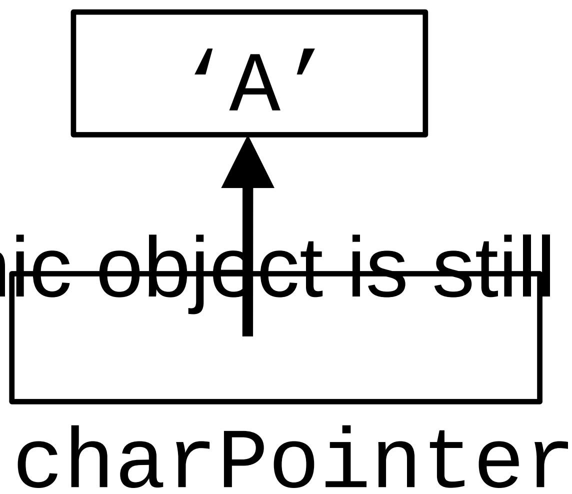
But …

dynamic memory objects still are marked as in use, but cannot be accessed when the function ends!!

Called a *Memory leak*

Dynamic objects stick around

```
{char* charPointer;
 charPointer = new char;
 *charPointer = 'A';}
```



During the function

After the function, local variable charPointer disappears

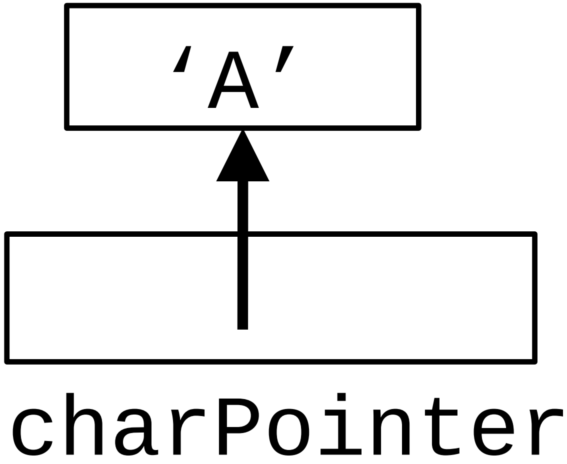
Dynamic object is still there!!

Removing dynamic object

To remove a dynamic object,

```
delete charPointer;
```

This removes the dynamic object charPointer is pointing to, but



`charPointer` is still around and still has the old address.

Removing dynamic object

What happens if you tried this?

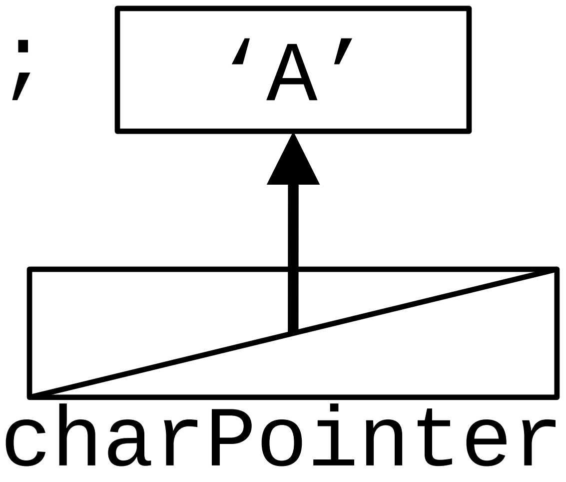
```
delete charPointer;
*charPointer = ‘B’;
```

Run-time error!!

Removing dynamic object

If you are not going to store another address in the pointer immediately after using `delete`, put in NULL.

```
delete charPointer;
charPointer = NULL;
```



References

- Two names for the same memory location
- The reference is an “alias”

References

```
string author = “Samuel Clemens”;
string& penName = author;
penName = “Mark Twain”;
cout << author;
```

What is output?

“Mark Twain”

“Mark Twain”

`author/penName`

Call by value/reference

- How to pass data between functions?
- Default: call by value
  - – Creates a new variable and copies data from caller to callee
- Call by reference (&)
  - – Copying data is slow; not always needed
  - – Callee may want to change caller’s data (sneaky!)

Call by Value

```
#include <iostream>
using namespace std;
void swap(int x, int y){
   int temp;
  temp = x;
  x = y;
  y = temp;
   return;
}
int main() {
    int a = 7;
    int b = 10;
    cout << "Before Swap " << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    swap(a, b);
    cout << "After Swap " << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}
```

Call by Reference

```
#include <iostream>
using namespace std;
void swap(int &x, int &y){
   int temp;
  temp = x;
  x = y;
  y = temp;
   return;
}
int main() {
    int a = 7;
    int b = 10;
    cout << "Before Swap " << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    swap(a, b);
    cout << "After Swap " << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}
```

Call by Pointer

```
#include <iostream>
using namespace std;
void swap(int* x, int* y){
  int temp;
 temp = *x;
 *x = *y;
 *y = temp;
  return;
}
int main() {
   int a = 7;
   int b = 10;
   cout << "Before Swap " << endl;
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
   swap(&a, &b);
   cout << "After Swap " << endl;
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
   return 0;
}
```

Points to think about

- What is the difference between Call by Reference and Call by Pointer ?

References

- M. Molodowitch
- CSUF CPSC 131 Slides: Pointers and Dynamic Variables, Dr. Anand Panangadan

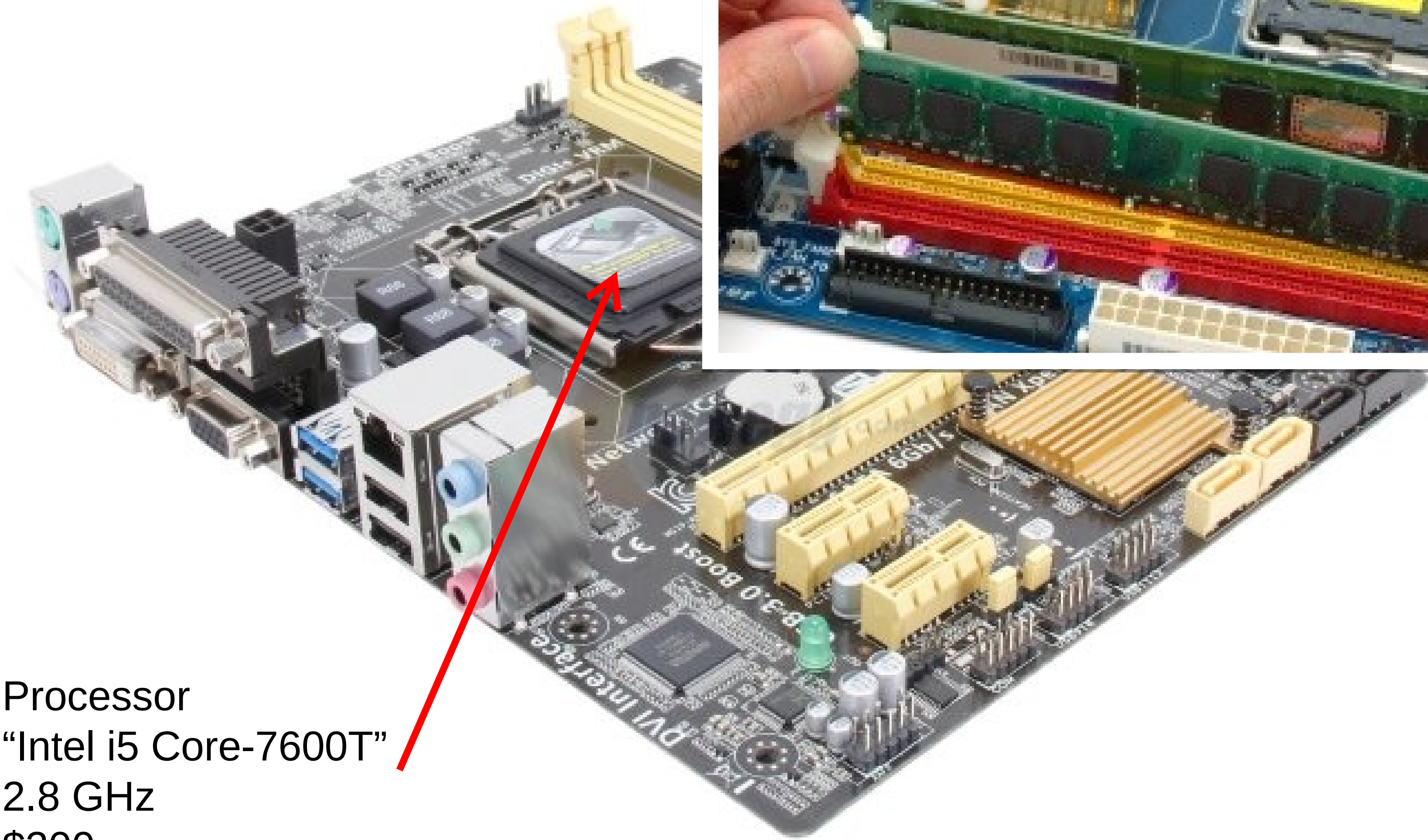
CPSC 131
Data Structures Concepts

Dr. Anand Panangadan
apanangadan@fullerton.edu

What we will cover today

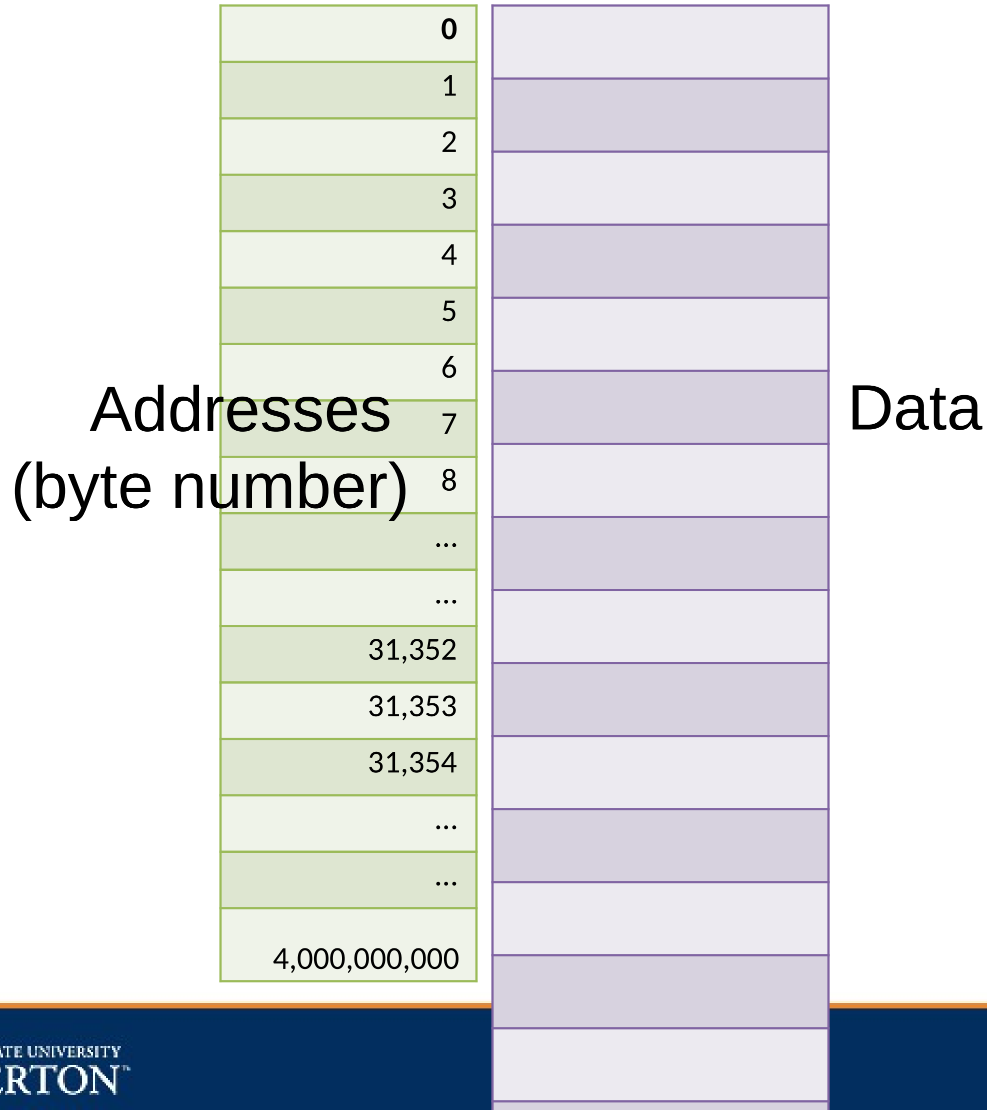
- review pointers, arrays, references, dynamic memory

Main memory
“Kingston DDR 3 DIMM”
8 GB
$80

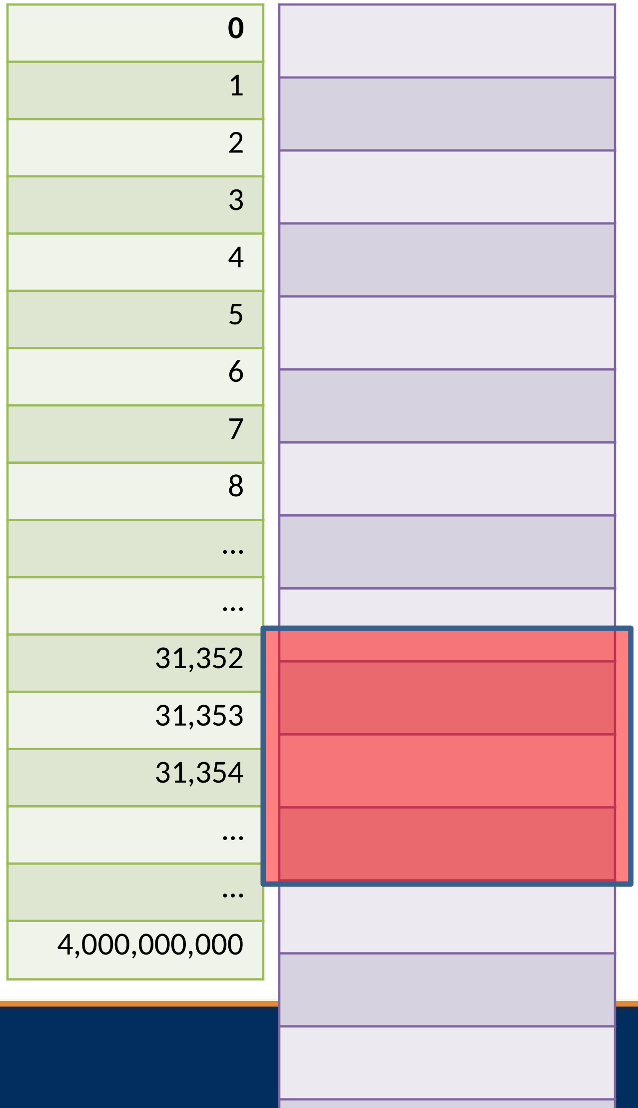


Processor
“Intel i5 Core-7600T”
2.8 GHz
$200

Data and Memory



Data and Memory




```
int n = 5;
n++;
```

Pointers

Working with pointers is not that difficult

  if…

you can represent pointers with pictures

and …

can translate back and forth

  between pictures and code.

Pointers

Two different concepts

1. A variable
2. *Location* of the variable

Pointers are variables that store

memory addresses of other variables

Usually a pointer will hold addresses of variables for a specific data type:

Example: Addresses of integer variables only, not for variables of type double.

Pointer syntax

Pointer variable declaration

DataType   *   PointerVariableIdentifier;

```
int  *  intPointer;
```

means

`intPointer` holds addresses of variables of integer data type.

Picture of a pointer variable

As with all other variables, a pointer variable is represented by a box labeled by the variable name.

```
int * intPointer;
```

intPointer

&: Address-of Operator

The address operator returns the address of the variable that follows it.

Ex. If `number` is an integer variable,

`&number` will return the actual memory address of the variable `number`.

So

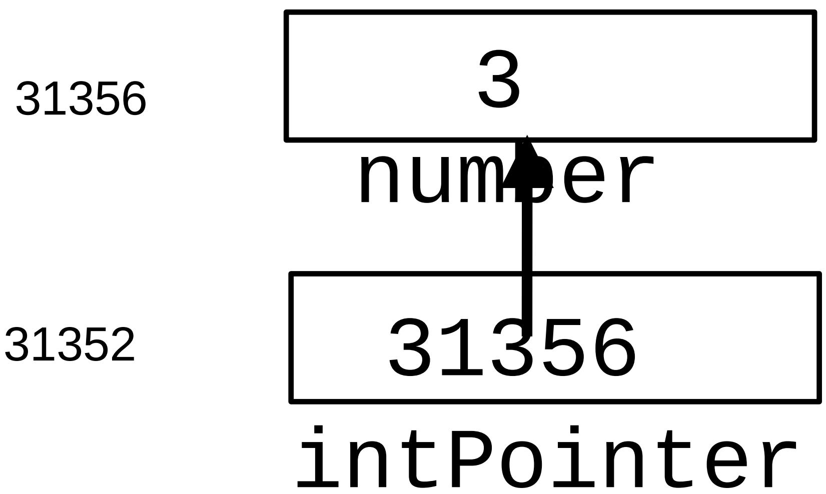
```
intPointer = &number;
```

means that `intPointer` will hold the address of `number`.

`Read, “intPointer is equal to the address of number”`

Picture of previous code

```
int number = 3;
int * intPointer;
intPointer = &number;
```



Assume memory location at address 31356 is reserved for number

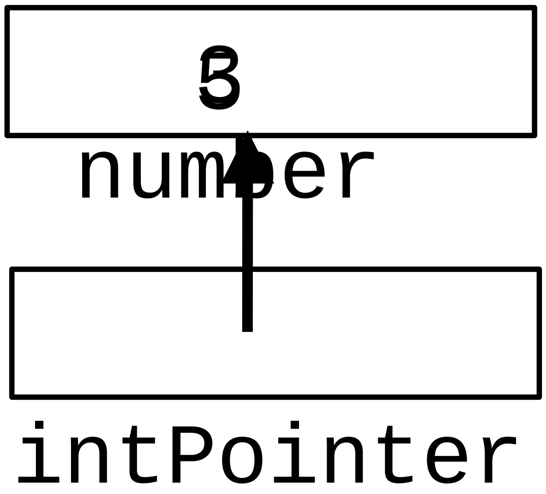
Assume memory location at address 31352 is reserved for `intPointer`

Since we don't know the actual addresses, we usually depict the value stored in a pointer with an arrow pointing to the variable whose address it stores.

Can also say "`intPointer` **points** to the variable `number`."

*: Dereferencing Operator

***** used with a pointer variable is an operator that returns the variable the pointer points to, so in the previous picture:



`*intPointer` is the same as `number` and

`*intPointer = 5;`
will mean

Dynamic Memory

We can create new objects while the program is running by using pointers.

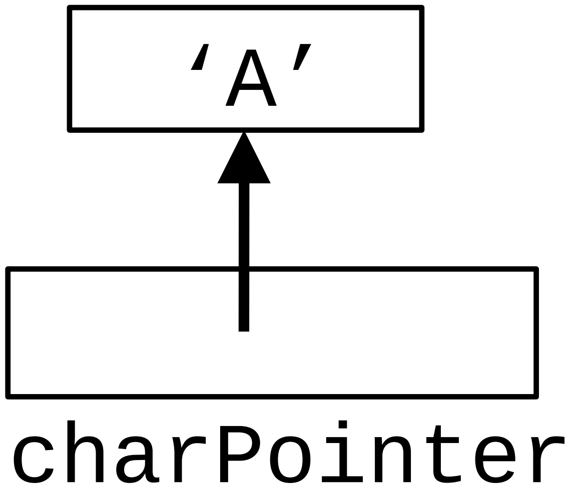
```
intPointer = new int;
```

We need a pointer variable to hold the address of the newly created dynamic object because we don't know where in memory the new object will be until the program is running!

Also called Heap memory or Free store

Picture of creating object in dynamic memory

```
char * charPointer;
charPointer = new char;
*charPointer = 'A';
```



Removing a dynamically allocated object

Static variables disappear when the function it is inside ends.

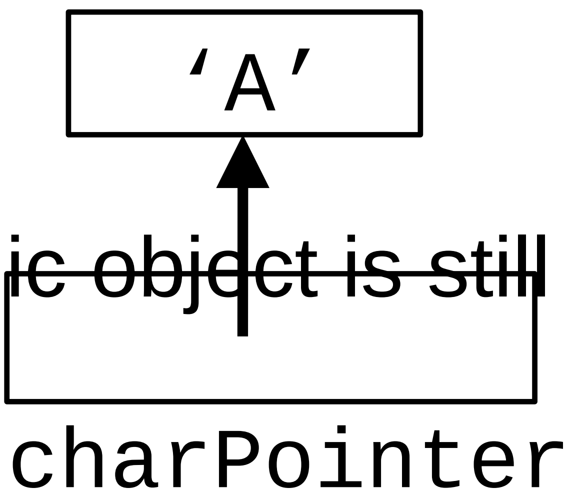
But …

dynamic memory objects still are marked as in use, but cannot be accessed when the function ends!!

Called a *Memory leak*

Dynamic objects stick around

```
{char * charPointer;
 charPointer = new char;
 *charPointer = ‘A’;}
```



Dynamic object is still there!!

During the function

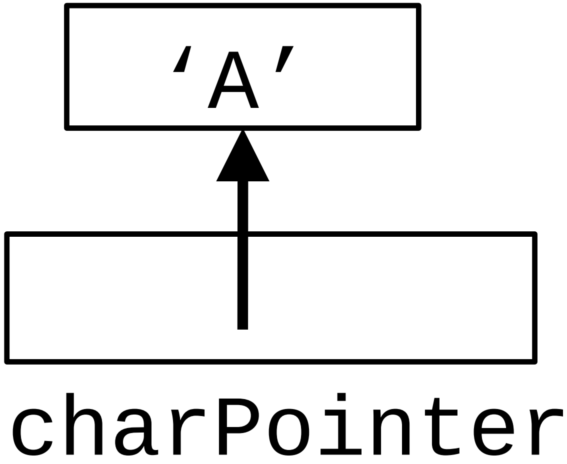
After the function, local variable charPointer disappears

Removing dynamic object

To remove a dynamic object,

```
delete charPointer;
```

This removes the dynamic object charPointer is pointing to, but



`charPointer` is still around and still has the old address.

Removing dynamic object

What happens if you tried this?

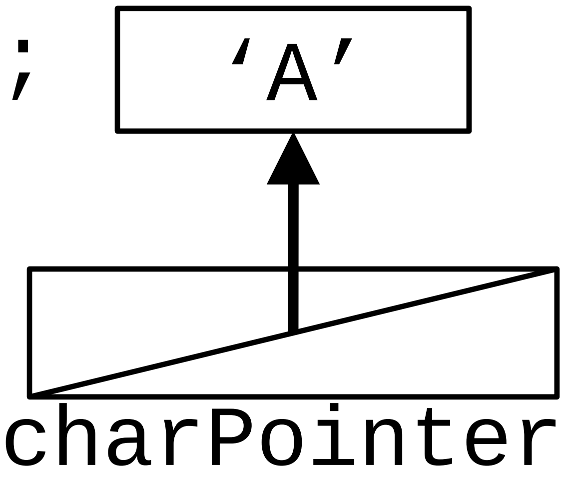
```
delete charPointer;
*charPointer = ‘B’;
```

Run-time error!!

Removing dynamic object

If you are not going to store another address in the pointer immediately after using `delete`, put in NULL.

```
delete charPointer;
charPointer = NULL;
```



References

- Two names for the same memory location
- The reference is an “alias”

References

```
string author = “Samuel Clemens”;
string& penName = author;
penName = “Mark Twain”;
cout << author;
```

What is output?

“Mark Twain”

“Mark Twain”

`author/penName`

Call by value/reference

- How to pass data between functions?
- Default: call by value
  - – Creates a new variable and copies data from caller to callee
- Call by reference (&)
  - – Copying data is slow; not always needed
  - – Callee may want to change caller's data (sneaky!)

perro

perrito

Luis

Luisito

Call by value/reference

- “Use references when you can, and pointers when you have to”

1. use pass by value if the type is small (4 Bytes) and don't want to have it changed after the return of the call
2. use pass by reference to const if the type is larger and you don't want to have it changed after the return of the call
3. use pass by reference if the parameter can't be NULL
4. use a pointer otherwise

Reference vs Pointer

- A pointer can be re-assigned any number of times while a reference cannot be re-assigned after binding.
- Pointers can point nowhere (NULL), whereas a reference always refers to an object.

Reference vs Pointer

"Use references when you can, and pointers when you have to"

- References can be used in function parameters and return types
- Pointers can be used for implementing algorithms and data structures.

1.1 Why pointers: A list example

A challenging and yet powerful programming construct is something called a *pointer*. This section describes one of many situations where pointers are useful.

A vector (or array) stores a list of items in contiguous memory locations. Storing in contiguous locations enables immediate access to any element of vector v by using `v.at(i)` (or `v[i]`), because the compiler just adds i to the starting address of v to access the element at index i. However, inserting an item requires making room by shifting higher-indexed items. Similarly, erasing an item requires shifting higher-indexed items to fill the gap. Shifting each item requires a few processor instructions. For vectors with thousands of elements, a single call to insert() or erase() can require thousands of instructions, so a program with many inserts or erases on large vectors may run very slowly, what we call the ***vector insert/erase performance problem***.

PARTICIPATION ACTIVITY 1.1.1: Vector insert performance problem.

Animation captions:

1. Inserting an item at a specific location in a vector requires making room for the item by shif higher-indexed items.

The following program demonstrates. The user inputs a vector size, and a number of operations (numOps) to perform. The program then resizes the vector, writes a value to each element, does numOps push_backs, does numOps inserts, and does numOps erases. The `<< flush` forces cout to ***flush*** any characters in its buffer to the screen before doing each task, otherwise the characters may be held in the buffer until after a later task completes. Running the program for vectorSize of 100000 and numOps 40000 shows that the writes and push_backs execute fast, but the inserts and erases are noticeably slow.

Figure 1.1.1: Program illustrating that vector inserts and erases can be slow.

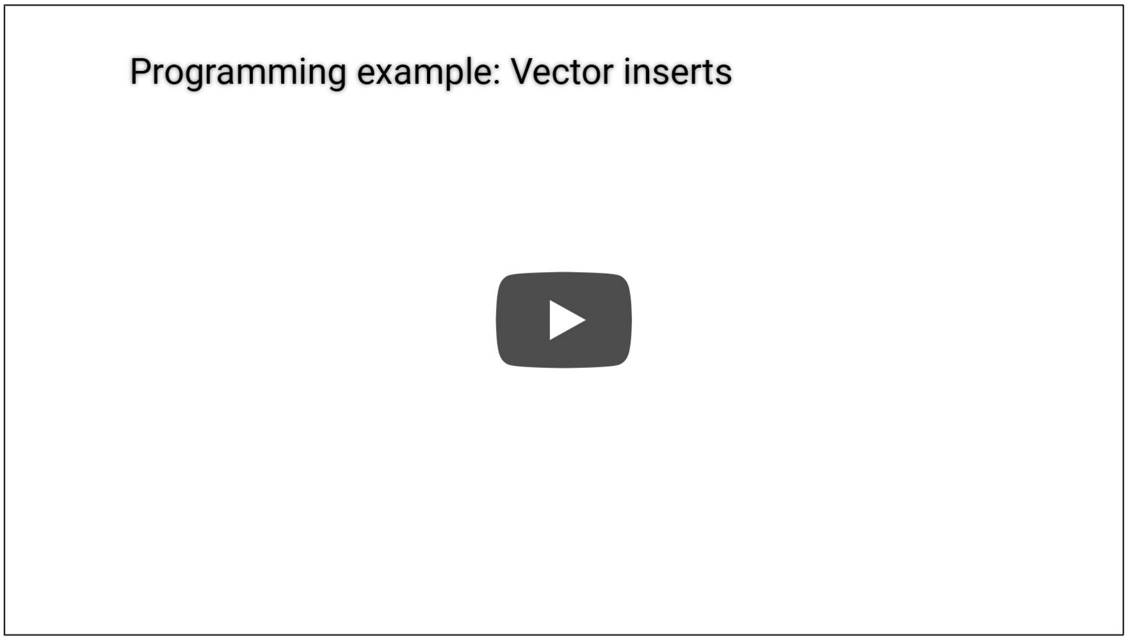
```
#include <iostream>
#include <vector>
using namespace std;

int main() {
   vector<int> tempValues; // Dummy vector to demo
vector ops
   int vectorSize;         // User defined size
   int numOps;             // Number of operations to
perform
   int i;                  // Loop index

   cout << "Enter initial vector size: ";
   cin >> vectorSize;

   cout << "Enter number of operations: ";
   cin >> numOps;

   cout << "  Resizing vector..." << flush;

   tempValues.resize(vectorSize);

   cout << "done." << endl;
   cout << "  Writing to each element..." << flush;

   for (i = 0; i < vectorSize; ++i) {
      tempValues.at(i) = 777; // Any value
   }

   cout << "done." << endl;
   cout << "  Doing " << numOps << " pushbacks..." <<
flush;

   for (i = 0; i < numOps; ++i) {
      tempValues.push_back(888); // Any value
   }

   cout << "done." << endl;
   cout << "  Doing " << numOps << " inserts..." <<
flush;

   for (i = 0; i < numOps; ++i) {
      tempValues.insert(tempValues.begin() + 0, 444);
   }
   cout << "done." << endl;
   cout << "  Doing " << numOps << " erases..." <<
flush;

   for (i = 0; i < numOps; ++i) {
      tempValues.erase(tempValues.begin() + 0);
   }

   cout << "done." << endl;

   return 0;
}
```

```
Enter initial vector size: 100000
Enter number of operations: 40000
  Resizing vector...done.
(fast)
  Writing to each element...done.
(fast)
  Doing 40000 pushbacks...done.
(fast)
  Doing 40000 inserts...done.
(SLOW)
  Doing 40000 erases...done.
(SLOW)
```


The push_backs are fast because they do not involve any shifting of elements, whereas each insert requires 100,000 elements to be shifted, one at a time. 40,000 inserts thus requires 4,000,000,000 shifts.

The video shows the program running for different vector sizes and number of operations; notice that for large values, the resize, writes, and push_backs all run quickly, but the inserts and erases take a noticeably long time.

Video 1.1.1: Vector inserts.



PARTICIPATION ACTIVITY 1.1.2: Vector insert/erase problem.

For each operation, how many elements must be shifted? Assume no new memory needs to be allocated. Questions are for vectors, but apply to arrays too.

1) Append an item to the end of a 999-element vector (e.g., using push_back()).

Check Show answer

2) Insert an item at the front of a 999-element vector.

Check Show answer

3) Delete an item from the end of a 999-element vector.

Check Show answer

4) Delete an item from the front of a 999-element vector.

Check Show answer

One way to make inserts or erases faster is to use a different approach for storing a list of items. The approach does not use contiguous memory locations. Instead, each item contains a "pointer" to the next item's location in memory, as well as the data being stored. Thus, inserting a new item B between existing items A and C just requires changing A to point to B's memory location, and B to point to C's location, as shown in the following animation.

PARTICIPATION ACTIVITY 1.1.3: A list avoids the shifting problem.

Animation captions:

1. List's first two items initially: (A, C, ...). Item A points to the next item at location 88. Item C points to next item at location 113 (not shown).
2. To insert new item B after item A, the new item B is first added to memory at location 90.
3. Item A is updated to point to location 90. Item B is set to point to location 88. New list is (A, ...). No shifting of items after C was required.

The initial list contains an item with data A followed by an item with C. Inserting item B did not require C to be shifted.

A ***linked list*** is a list wherein each item contains not just data but also a pointer—a *link*—to the next item in the list. Comparing vectors and linked lists:

- *Vector*: Stores items in contiguous memory locations. Supports quick access to i'th element via `v.at(i)`, but may be slow for inserts or deletes on large lists due to necessary shifting of elements.
- *Linked list*: Stores each item anywhere in memory, with each item pointing to the next item in the list. Supports fast inserts or deletes, but access to i'th element may be slow as the list must be traversed from the first item to the i'th item. Also uses more memory

due to storing a link for each item.

A vector/array is like people ordered by their seat in a theater row; if you want to insert yourself between two adjacent people, other people have to shift over to make room. A linked list is like people ordered by holding hands; if you want to insert yourself between two people, only those two people have to change hands, and nobody else is affected.

PARTICIPATION ACTIVITY 1.1.4: Linked list inserts/deletes using pointers.

1) Appending an item at the end of a 999-item linked list requires how many items to be shifted?

Check Show answer

2) Inserting a new item between the 10th and 11th items of a 999-item linked list will require a few pointer changes. In addition, how many shifts will be required?

Check Show answer

3) Finding the 500th item in a 999-item linked list requires visiting how many items? Correct answer is one of 0, 1, 500, and 999.

Check Show answer

Exploring further:

- Pointers tutorial from cplusplus.com
- Pointers article from cplusplus.com

How was this section? Provide feedback

1.2 Pointer basics

A ***pointer*** is a variable that contains a memory address, rather than containing data like most variables introduced earlier. The following program introduces pointers via example:

Figure 1.2.1: Introducing pointers via a simple example.

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

int main() {
   int usrInt;      // User defined int value
   int* myPtr = nullptr; // Pointer to the user defined int value

   // Prompt user for input
   cout << "Enter any number: ";
   cin >> usrInt;

   // Output int value and address
   cout << "We wrote your number into variable usrInt." << endl;
   cout << "The content of usrInt is: " << usrInt << "." << endl;
   cout << "usrInt's memory address is: " << &usrInt << "." << endl;
   cout << endl << "We can store that address into pointer variable myPtr."
        << endl;

   // Grab address storing user value
   myPtr = &usrInt;

   // Output pointer value and value at pointer address
   cout << "The content of myPtr is: " << myPtr << "." << endl;
   cout << "The content of what myPtr points to is: "
        << *myPtr << "." << endl;

   return 0;
}
```

```
Enter any number: 555
We wrote your number into variable usrInt.
The content of usrInt is: 555.
usrInt's memory address is: 0x7fff5fbff718.

We can store that address into pointer variable myPtr.
The content of myPtr is: 0x7fff5fbff718.
The content of what myPtr points to is: 555.
```


The example demonstrates key aspects of working with pointers:

- *Appending * after a data type* in a variable declaration declares a pointer variable, as in