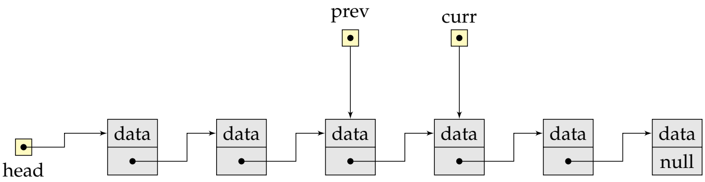
CSCI 2270: Data Structures

Lecture 06: Parameter Passing, Pointers, and References

Ashutosh Trivedi



Department of Computer Science
University of Colorado Boulder

Arrays, Pointers, and References

Pass-by-Value, Pass-by-Pointers, and Pass-by-Reference

What’s an Array?

What’s an Array?

An array is a contiguous allocation of elements of same type.

What’s an Array?

An array is a contiguous allocation of elements of same type.

There are two ways to allocate an array:

- **Static Allocation**:

```
int x[5];
float y[] = {33, 44, 55};
```

What’s an Array?

An array is a contiguous allocation of elements of same type.

There are two ways to allocate an array:

- **Static Allocation**:

```
int x[5];
float y[] = {33, 44, 55};
```

- **Dynamic Allocation**:

```
int* x = new int[N];
delete[] x;
```

- When to use static vs dynamic allocation?

What's an Array?

An array is a contiguous allocation of elements of same type.

There are two ways to allocate an array:

- **Static Allocation**:

```
int x[5];
float y[] = {33, 44, 55};
```

- **Dynamic Allocation**:

```
int* x = new int[N];
delete[] x;
```

- When to use static vs dynamic allocation?
  1. When the size to be allocated is fixed.
  2. When the size to be allocated is changing.
  3. Other concerns: large array, need to move around between methods, and so on.
  4. You are responsible for managing heap-allocated memory.

What is a Pointer?

What is a Pointer?

1. *A pointer is a data type that “points to” another value stored in memory.*

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*
2. *A pointer is a variable that holds the "memory address" where a value lives.*

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*
2. *A pointer is a variable that holds the "memory address" where a value lives.*

```
int x = 33;
```

Type	Name	Value	Address
int	x	33	0001
			0002
			0003
			0004
			0005

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*
2. *A pointer is a variable that holds the "memory address" where a value lives.*

```
int y[2] = {11, 14};
```

Type	Name	Value	Address
int	x	33	0001
int	y[0]	11	0002
int	y[1]	14	0003
			0004
			0005

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*
2. *A pointer is a variable that holds the "memory address" where a value lives.*

```
int* p = &x;
```

Type	Name	Value	Address
int	x	33	0001
int	y[0]	11	0002
int	y[1]	14	0003
			0004
int*	p	0001	0005

Arrays are like pointers

- In C++, arrays and pointers are intimately connected!
- The name of the array can be used as a pointer to its initial element.

```
// program32-m.cpp
#include<iostream>
#include<cassert>
void foo();
int main(int argc, char* argv[])  {
  int v[] = {1, 2, 3, 4};
  int *p1 = v; // v is a point to the 0-th element of the vector v
  int *p2 = v+1; // v+i is a point to the i-th element of the vector v
  int p3 = v[2];
  int p4 = *(v+2); // v[i] is a shorthand for  *(v+i)

  assert(p3 == p4 && "this should never execute");

  return 0;
}
```

What’s a reference?

- A *reference* is an alternative name for an object.
- The key use of a reference is in specifying arguments and return values.
- To ensure that a reference is a name for something, we must initialize it.
- Once assigned, it can not be reassigned to another object.

```cpp
// program33-m.cpp
#include<iostream>
using namespace std;

int main(int argc, char* argv[]) {
  int x = 7;
  //int& y; // error
  //int& y = 7; // error
  int& y = x; // once and for all bound to x

  cout << "x = " << x << " Add of x:" << &x << endl;
  cout << "y = " << x << " Add of y: " << &y << endl;

  y++;

  cout << "x = " << x << " Add of x:" << &x << endl;
  cout << "y = " << x << " Add of y: " << &y << endl;

  return 0;
}
```

References (A Rose by another name!)

- Function arguments can be references themselves and they get initialized at function call.

```
// program33-m2.cpp
#include<iostream>
using namespace std;

void foo(int& y)
{
  y = 20;
}

int main(int argc, char* argv[]) {
  int x = 7;
  cout << "x = " << x << " Add of x:" << &x << endl;

  foo(x);

  cout << "x = " << x << " Add of x:" << &x << endl;
  return 0;
}
```

Arrays, Pointers, and References

Pass-by-Value, Pass-by-Pointers, and Pass-by-Reference

Chatting across Stack frames: Pass by Value

```cpp
// program34.cpp -- pass by value
#include<iostream>
void swap_simple(int a, int b);
int main(int argc, char* argv[])
{
  int x = 5;
  int y = 7;

  std::cout << "x = " << x << " y = " << y << std::endl;
  swap_simple(x, y); // pass by value
  std::cout << "x = " << x << " y = " << y << std::endl;

  return 0;
}
void swap_simple(int a, int b) {
  int c;
  c = a;
  a = b;
  b = c;
}
```

Chatting across Stack frames: Pass by Pointers

```
// program35.cpp -- pass by pointers
#include<iostream>
void swap_simple(int *a, int *b);
int main(int argc, char* argv[])
{
  int x = 5;
  int y = 7;

  std::cout << "x = " << x << " y = " << y << std::endl;
  swap_simple(&x, &y); // pass by pointers
  std::cout << "x = " << x << " y = " << y << std::endl;

  return 0;
}
void swap_simple(int *a, int *b) {
  int z;
  z = *a;
  *a = *b;
  *b = z;
}
```

Chatting across Stack frames: Pass by Reference

```cpp
// program36.cpp-- pass by reference
#include<iostream>
void swap_simple(int& a, int& b);
int main(int argc, char* argv[])
{
  int x = 5;
  int y = 7;

  std::cout << "x = " << x << " y = " << y << std::endl;
  swap_simple(x, y); // pass by reference
  std::cout << "x = " << x << " y = " << y << std::endl;

  return 0;
}
void swap_simple(int& a, int& b) {
  int c;
  c = a;
  a = b;
  b = c;
}
```

Distinguish between pointers and references

- Pointers and reference look different enough (pointers use “*” and “->” operators and references use .).
- But they seem to do the same thing: They let you refer to other objects indirectly.
- So when to use pointer and when reference?

Distinguish between pointers and references

- Pointers and reference look different enough (pointers use “*” and “->” operators and references use .).
- But they seem to do the same thing: They let you refer to other objects indirectly.
- So when to use pointer and when reference?
  1. Note: There is no such thing as a null reference!

Distinguish between pointers and references

- Pointers and reference look different enough (pointers use “*” and “->” operators and references use .).
- But they seem to do the same thing: They let you refer to other objects indirectly.
- So when to use pointer and when reference?
  1. Note: There is no such thing as a null reference!
  2. Note: Pointers may be reassigned to different objects!
  3. Note: Reference based access may be faster than pointers.

Best practices

1. *Pass large objects only by reference or by pointers.*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*
4. *Free the memory that doesn't spark joy!*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*
4. *Free the memory that doesn't spark joy!*
5. *Be extremely careful in using references! Use it for speed and memory!*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*
4. *Free the memory that doesn't spark joy!*
5. *Be extremely careful in using references! Use it for speed and memory!*
6. *If the invoked method is not supposed to change the value, use the "const".*

   ```
   void Func3(const int& x);// pass by const reference
   ```

   This would be used to avoid the overhead of making a copy, but still prevent the data from being changed.
7. *Be mindful that arrays can not be passed by value!*
8. *Be mindful when returning references and pointers*

   ```
   int& doit () {
   int x = 0;
   return x;
   ```

1. What is wrong with this code?

```cpp
// program40.cpp
#include<iostream>
int* foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int *res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << *res << std::endl;
    *res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << *res << std::endl;
    return 0;
  }
}


int* foo(int x) {
  int z = x*x;
  std::cout << &z << std::endl;
  return &z;
}
```

2. What is wrong with this code?

```cpp
// program41.cpp
#include<iostream>
int& foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << res << std::endl;
    res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << res << std::endl;

    return 0;
  }
}

int& foo(int x) {
  int z = x*x;
  return z;
}
```

2. What is wrong with this code?

```cpp
// program41.cpp
#include<iostream>
int& foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << res << std::endl;
    res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << res << std::endl;

    return 0;
  }
}

int& foo(int x) {
  int z = x*x;
  return z;
}
```

Be careful when returning references or pointers. (use heap memory or `static` keyword)

3. What is wrong with this code?

```cpp
// program42.cpp
#include<iostream>
int* foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int *res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << *res << std::endl;
    *res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << *res << std::endl;
    // Some other computation that uses "res"
    // Some other computation that does not use "res"
    return 0;
  }
}

int* foo(int x) {
  int *z = new int(x*x);
  return z;
}
```

3. Fixed!

```cpp
// program43.cpp
#include<iostream>
int* foo(int x);
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cout << "provide a number as an argument" << std::endl;
    return -1;
  }
  else {
    int *res = foo(atoi(argv[1]));
    std::cout << "The function returned: " << *res << std::endl;
    *res = 1234; // change the value stored at address pointed by res
    std::cout << "New value: " << *res << std::endl;
    // Some other computation that uses "res"
    delete res;
    res = 0;
    // Some other computation that does not use "res"
    return 0;
  }
}

int* foo(int x) {
  int *z = new int(x*x);
  return z;
}
```

4. What is wrong with this code?

```cpp
// program44.cpp
#include<iostream>
int* foo(int  x);
int main(int argc, char* argv[]) {
  if (argc == 2) {
    int* res = foo(atoi(argv[1]));
    // Some other computation that uses "res"
    delete res;
    res = 0;
    // Some other computation that does not use "res"
  }
    return 0;
}
int* foo(int x) {
  int* res = new int[x];
  return res;
}
```

4. Fixed

```
// program45.cpp
#include<iostream>
int* foo(int  x);
int main(int argc, char* argv[]) {
  if (argc == 2) {
    int* res = foo(atoi(argv[1]));
    // Some other computation that uses "res"
    delete[] res;
    res = 0;
    // Some other computation that does not use "res"
  }
    return 0;
}
int* foo(int x) {
  int* res = new int[x];
  return res;
}
```

4. Fixed

```
// program45.cpp
#include<iostream>
int* foo(int  x);
int main(int argc, char* argv[]) {
  if (argc == 2) {
    int* res = foo(atoi(argv[1]));
    // Some other computation that uses "res"
    delete[] res;
    res = 0;
    // Some other computation that does not use "res"
  }
    return 0;
}
int* foo(int x) {
  int* res = new int[x];
  return res;
}
```

Use the same form in corresponding uses of new and delete.

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*
4. *Free the memory that has served its purpose.*
5. *Be extremely careful in using references! Use it for speed and memory!*
6. *If the invoked method is not supposed to change the value, use the "const".*

   ```
   void Func3(const int& x);// pass by const reference
   ```

   This would be used to avoid the overhead of making a copy, but still prevent the data from being changed.
7. *Be mindful that arrays can not be passed by value.*
8. *Be mindful when returning references and pointers.*
9. *Use the same form in corresponding uses of new and delete.*

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.
- It means that the invoked function can change the values stored at individual indices.

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.
- It means that the invoked function can change the values stored at individual indices.
- However, if you need to change structure of the array itself, it can not be done if you are passing by value! Why?

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.
- It means that the invoked function can change the values stored at individual indices.
- However, if you need to change structure of the array itself, it can not be done if you are passing by value! Why?
- If you wish to modify the structure of the array, you need to pass it by pointers.

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.
- It means that the invoked function can change the values stored at individual indices.
- However, if you need to change structure of the array itself, it can not be done if you are passing by value! Why?
- If you wish to modify the structure of the array, you need to pass it by pointers.

```cpp
void plusFour(int *array, int capacity) {
  for (int i=0; i< capacity; i++) {
    array[i] = array[i] + 4;
  }
}

int main(){
  int capacity = 10;
  int array[10];

  for (int i=0; i < capacity; i++) array[i] =1;

  plusFour(array, capacity);

  for (int i=0; i< capacity; i++) std::cout << array[i] << " ";
}
```

Arrays as Parameters (Contd.)

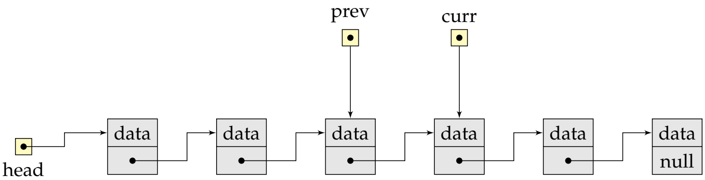
```cpp
void plusFour(int *array, int capacity) {
  int *newarray = new int[capacity];
  for (int i=0; i< capacity; i++) {
    newarray[i] = array[i] + 4;
  }

  array = newarray;
  std::cout << "Inside called function:" << std::endl;
  for (int i=0; i< capacity; i++) std::cout << array[i] << " ";

  std::cout<<std::endl;
}

int main(){
  int capacity = 10;
  int array[10];

  for (int i=0; i < capacity; i++) array[i] =1;

  plusFour(array, capacity);
  std::cout << "After returning:" << std::endl;
  for (int i=0; i< capacity; i++) std::cout << array[i] << " ";
}
```

Arrays as Parameters (Contd.)

```cpp
void plusFour(int** pArray, int capacity) {
  int *newarray = new int[capacity];
  for (int i=0; i< capacity; i++) {
    newarray[i] = (*pArray)[i] + 4;
  }

  delete[] (*pArray);
  *pArray = newarray;

  std::cout << "Inside called function:" << std::endl;
  for (int i=0; i< capacity; i++) std::cout << (*pArray)[i] << " ";

  std::cout<<std::endl;
}

int main(){
  int capacity = 10;
  int* array = new int[capacity];

  for (int i=0; i < capacity; i++) array[i] =1;

  plusFour(&array, capacity);
  std::cout << "After returning:" << std::endl;
  for (int i=0; i< capacity; i++) std::cout << array[i] << " ";
}
```

Why do we need pointers?

- To refer to memory allocated on the heap using “new”.
- Refer to and share large data-structures among functions (recall that passing-by-reference makes called function to copy the whole structure).
- A beautiful application in data-structure: Dynamically-expanding data-structures (Linked-Lists)!


prev
curr
data
data
data
data
data
data
null
head