lecture 6

- *C++ at a Glance*
  - *intro to pointers and references*
  - *comment on arrays and vector container*
  - *more on functions*

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

C++ at a glance

Pointers (1of2) (more coming on pointers in future lectures)

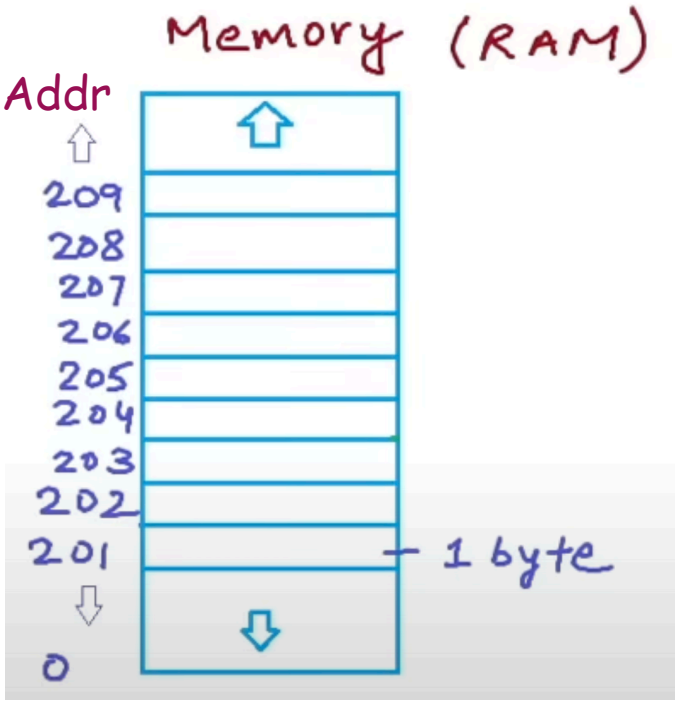
- variable that contains memory address of another variable, function, or object
  int p; //p* points to memory address that holds on integer value
  int i = 7;
  p = &i; // address of *i* assigned to *p; p* now points to memory address of *i*
  // &, address of operator
  **p =15; // *, dereference operator; go to address pointed to by p and assign the*
  // value in the address15

- copying a pointer is easy
  int q = p; //p* is type *int**, so is *q*

- pointers can point to pointers
  *int** r = &p;*

Kenneth J. Roche
University of Washington

C++ at a glance - pointers intro

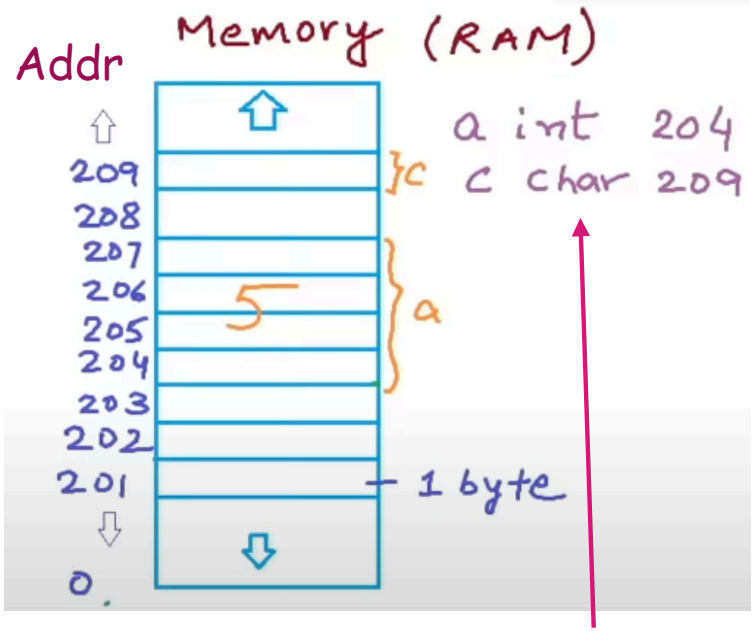



```
int a;
char c;
a = 5;
```

int - 4 bytes

char - 1 byte

float - 4 bytes



addr lookup table

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

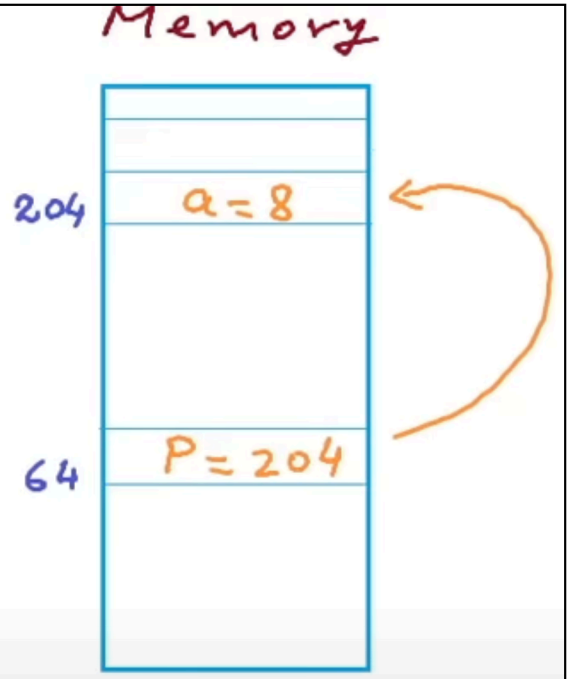
C++ at a glance - pointers intro

Pointers - variables that store address of another variable

P → address

*P → value at address

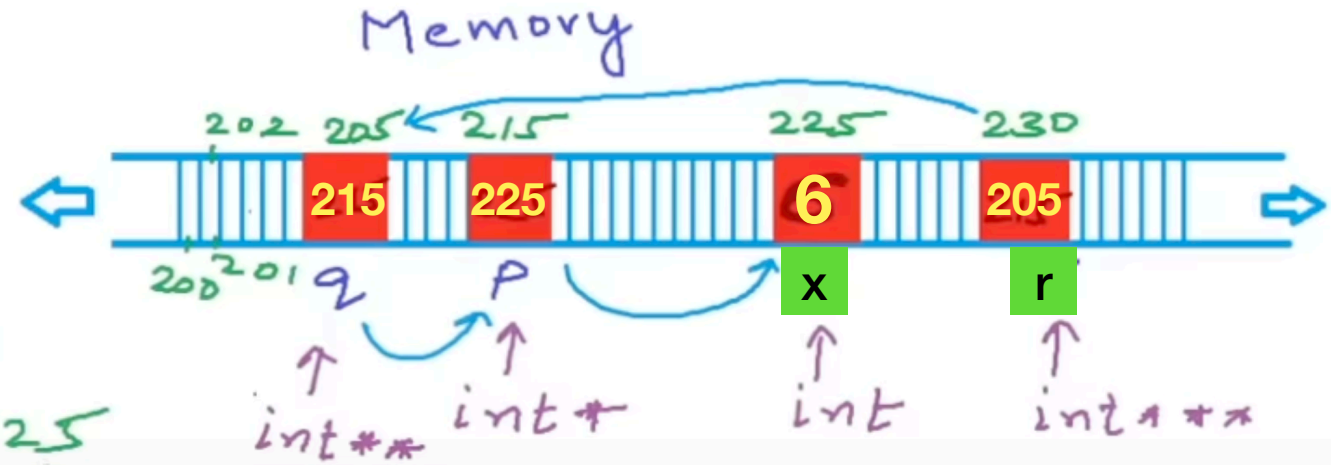
```
int a;
int *p;
p = &a;
a = 5;
Print p     // 204
Print &a    // 204
Print &p    // 64
print *p    // 5   ⇒ derefrencing
*p = 8
Print a     // 8
```



Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

C++ at a glance - pointers intro

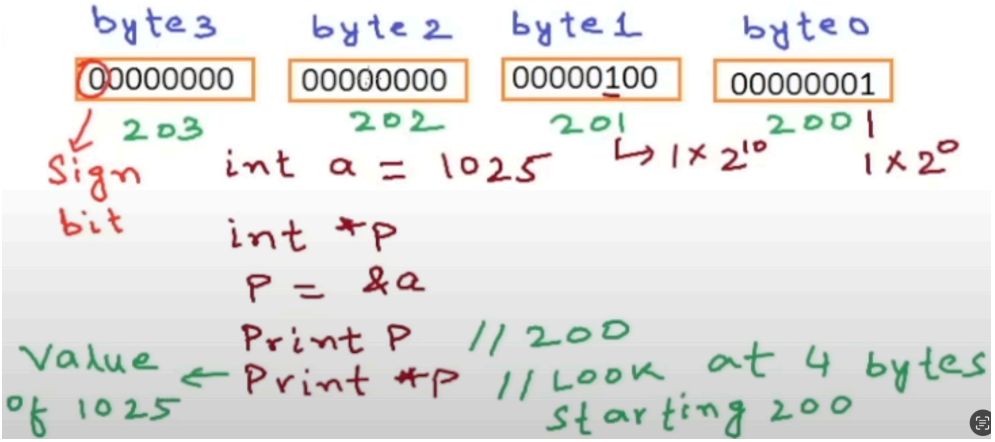
```
int x = 5;
int* p = &x;
*p = 6;
int** q = &p;
int*** r = &q;
printf("%d\n",*p); // 6
printf("%d\n",*q); // 225
printf("%d\n",*(*q)); // 6
```



Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

C++ at a glance - pointers intro

int – 4 bytes
char – 1 byte
float – 4 bytes

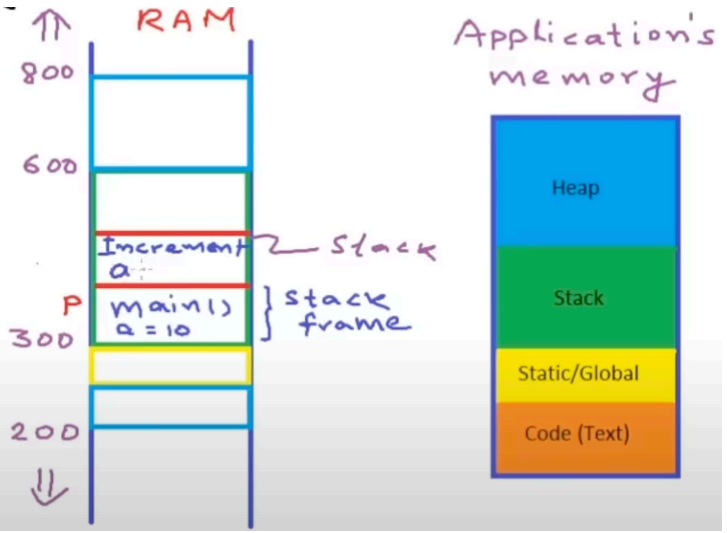



```
int a = 1025;
int *p;
p = &a;
printf("size of integer is %d bytes\n",sizeof(int));
printf("Address = %d, value = %d\n",p,*p);
printf("Address = %d, value = %d\n",p+1,*(p+1));
char *p0;
p0 = (char*)p; // typecasting
printf("size of char is %d bytes\n",sizeof(char));
printf("Address = %d, value = %d\n",p0,*p0);
printf("Address = %d, value = %d\n",p0+1,*(p0+1));
// 1025 = 00000000 00000000 00000100 00000001
```

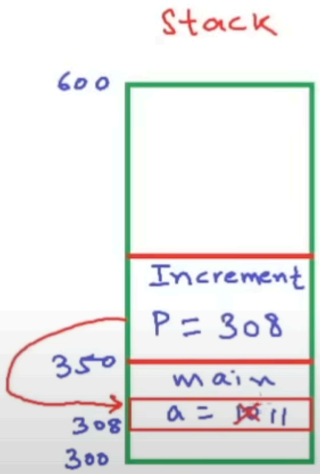
Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

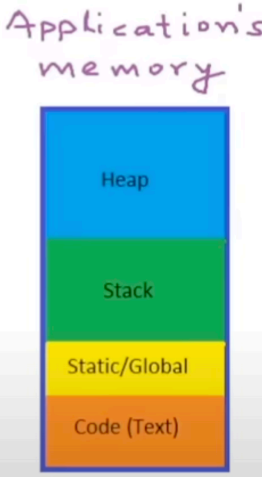
C++ at a glance - pointers intro

```
#include<stdio.h>
void Increment(int a)
{
    a = a+1;
}
int main()
{
    int a;
    a = 10;
    Increment(a);
    printf("a = %d",a);
}
```




```
#include<stdio.h>
void Increment(int *p)
{
    *p = (*p) + 1;
}
int main()
{
    int a;
    a = 10;
    Increment(&a);
    printf("a = %d",a);
}
```





Kenneth J. Roche
University of Washington

Arrays

- *int foo[3]; // declare integer array with 3 elements*
  - *foo[0] = 0; foo[1] =1; foo[2] = 2; // assignments*
  - *int foo[3] = {0,1,2}; //declare and assign at once*
  - *int foo[ ] = {0,1,2}; // implicit*
- *multi-dimensional*
  - *int foo[2][3] = {{0,1, 2}, {3, 4, 5}}; // declare integer array with 2 x 3 elements*
- *dynamically allocated (see pointers)*
- *vector container template class*
  - *resizable arrays*
  - *implicitly deallocated when out of scope*
  - *useful methods*

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

int main()
{
  vector<int> v;

  // Assign three elements with value two
  v.assign(3, 2); // [2, 2, 2]

  // Add 4 at last position
  v.push_back(4); // [2, 2, 2, 4]

  // Change first element
  v[0] = 1; // [1, 2, 2, 4]

  // Change second element (bound checked)
  v.at(2) = 3; // [1, 2, 3, 4]

  // Remove second element
  v.erase(v.begin()+1); // [1, 3, 4]

  // Remove last element
  v.pop_back(); // [1, 3]

  // Get vector length
  int len = v.size(); // 2

  // Print first and second elements
  cout << v.at(0) << v[1]; // "13"
}
```

Kenneth J. Roche
University of Washington

C++ at a glance - pointers intro

```
int A[] ={2,4,5,8,1};

int *p = A;
for(int i = 0;i<5;i++)
{
    printf("Address = %d\n",&A[i]);
    printf("Address = %d\n",A+i);
    printf("value = %d\n",A[i]);
    printf("value = %d\n",*(A+i));
}
```

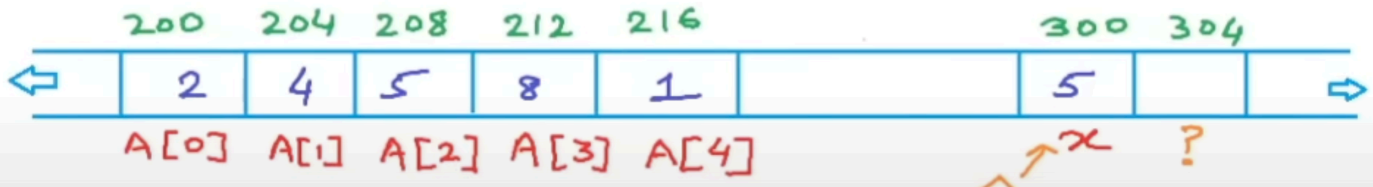
Pointers and Arrays

int A[5]

A[0]
A[1]
A[2]
A[3]
A[4]

int → 4 bytes
A → 5 × 4 bytes
= 20 bytes

int A[5]
int *p
P = A
Print A // 200
Print *A // 2



Element at index i –
Address – &A[i] or (A + i)
Value – A[i] or *(A+i)

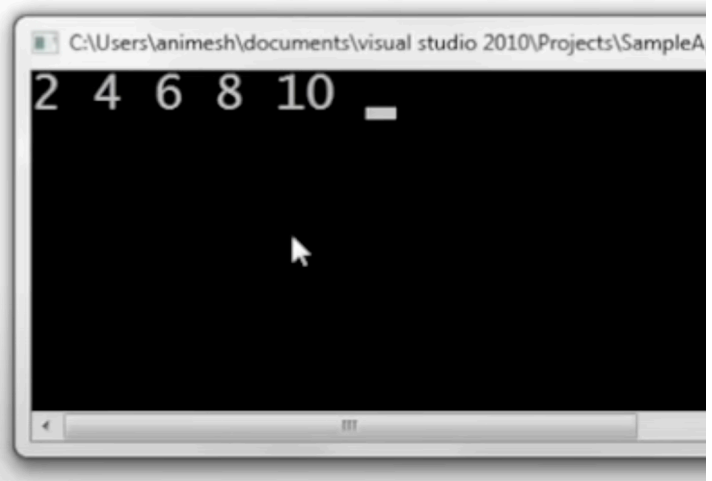
Print A+1 // 204
Print *(A+1) // 4

Kenneth J. Roche
University of Washington

AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

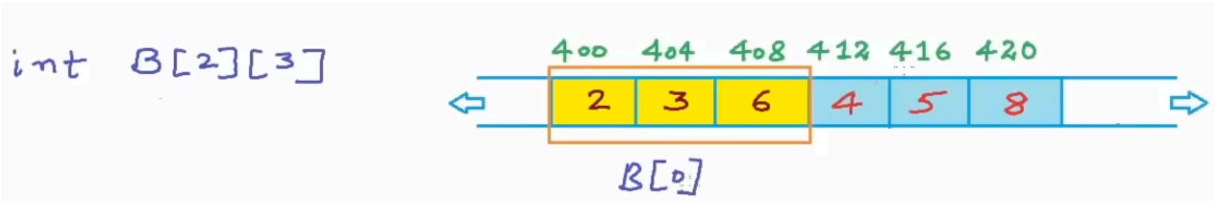
C++ at a glance - pointers intro

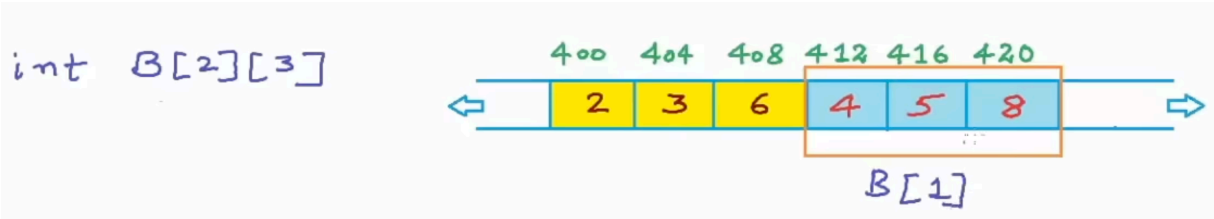
```
void Double(int* A, int size)// "int* A" or "int A[]"  ..it's the same..
{
    int i, sum = 0;
    for(i = 0;i< size;i++)
    {
        A[i] = 2*A[i];
    }
}
int main()
{
    int A[]= {1,2,3,4,5};
    int size = sizeof(A)/sizeof(A[0]);
    int i;
    Double(A,size);
    for(i = 0;i< size;i++)
    {
        printf("%d ",A[i]);
    }
}
```

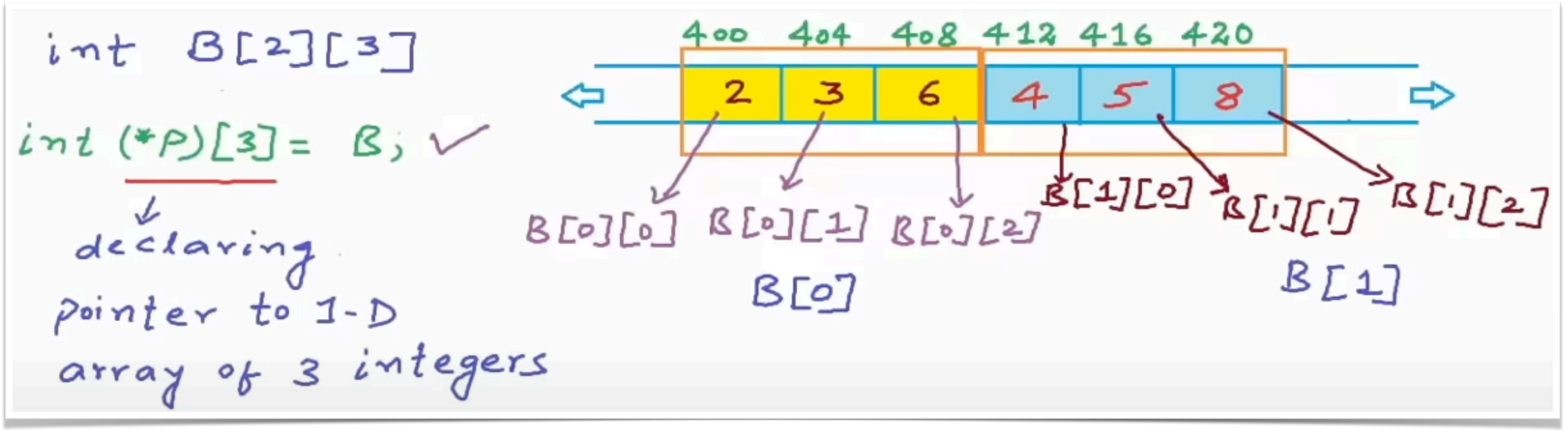


Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

C++ at a glance - pointers intro

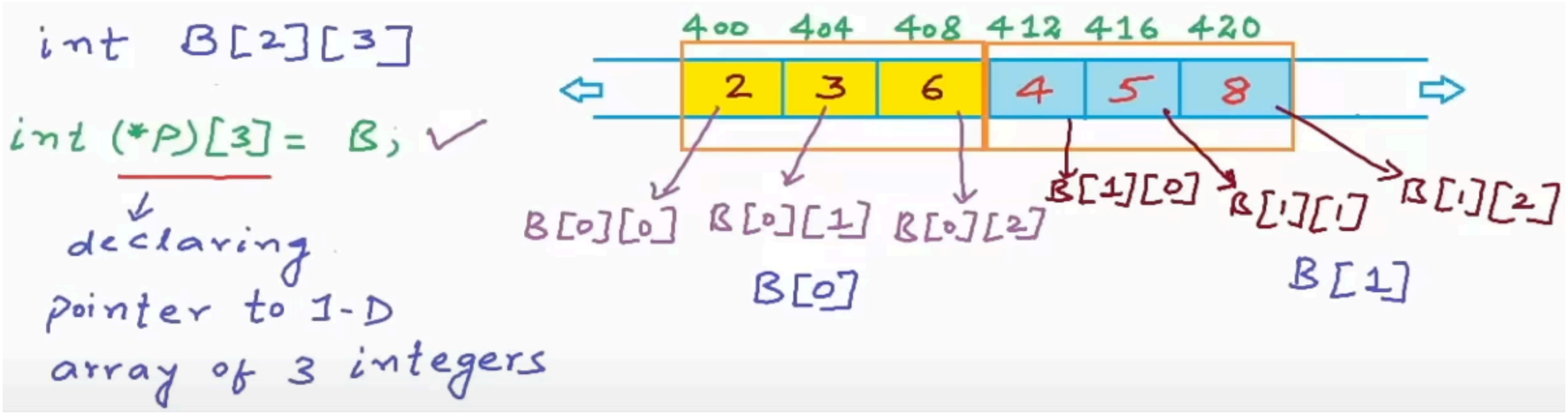


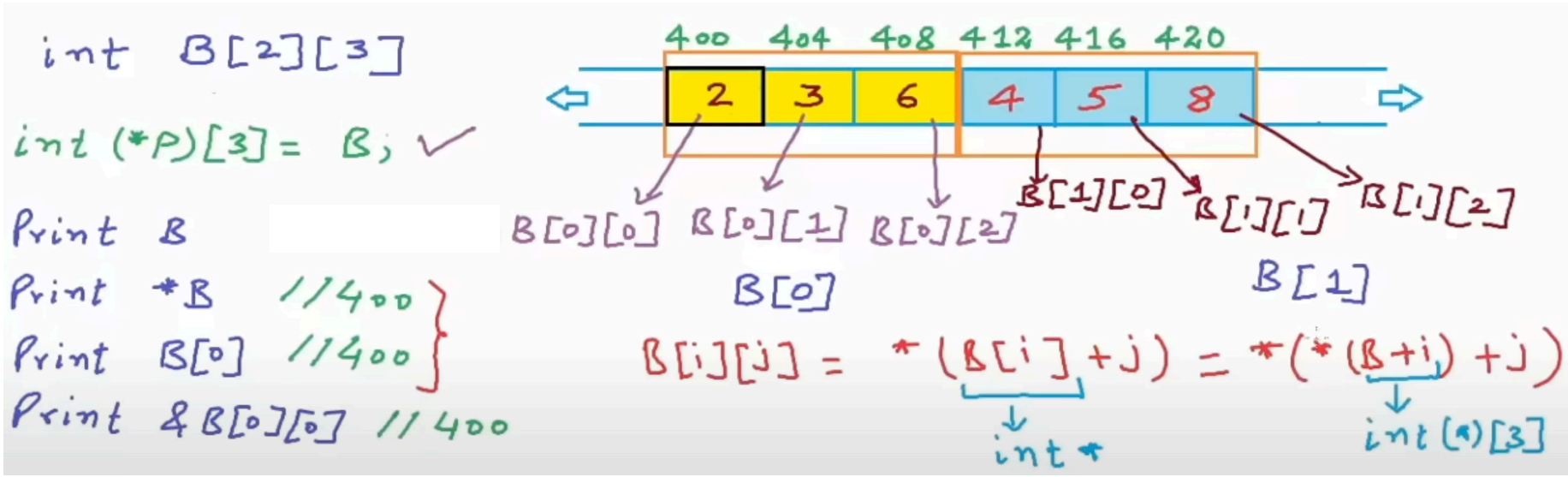




Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

C++ at a glance - pointers intro





Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

FUNCTIONs

```
void changei(int &i)
{
    i += 10;
}
```

```
int ix = 4;
std::cout << "init ix: " << ix << std::endl;
changei(ix);
std::cout << "new ix: " << ix << std::endl;
```

C++ at a glance

References

- *variable that can serve as an **alias** to a typed variable*

  int i = 7;

  int& r = i; // r references, i.e. is an alias of, *i*

  - must be initialized on declaration
  - can be used the same way the variable it was assigned
  - can never be used to reference another variable
  - does not have an address of its own

  r = 13; // now *i* has value *13*
- references should be used instead of pointers if pointer doesn't need to be reassigned
- rvalue reference (probably return to this but want to mention it)

  a temporary object that can be modified

  avoid unnecessary copying dealing with temporary objects (See Andrew's notes, http://thbecker.net/articles/rvalue_references/section_01.html)

C++ at a glance

Pointers (2of2)

- *int a[10];* // this is static allocation by declaration and stored on the **stack** of the process memory

- pointers allow dynamic allocation of memory from the **heap** of process memory
  int size = 10; //size may be decided at run time making malloc useful
  int p = new int[size];*
  delete[ ] p; // memory leaks are an issue … beware!
  p = nullptr; // mark pointer as unused to prevent misuse

- check for valid pointer
  *if (p != nullptr) {*p=30;}*
  *if (p) {*p=30;}* //same as above

C++ at a glance

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

void printSeparator(const string &label)
{
    cout << "\n=== " << label << " ===\n";
}

int main()
{
    printSeparator("Basic Pointer Example");

    int i = 7;
    int *p = &i; // p points to i
    cout << "Value of i: " << i << ", Address of i: " << &i << ", *p: " << *p << endl;
    cout << "Value of i: " << i << ", Address of i: " << &i << ", *p: " << *p << ", Address of p: " << &p << endl;
    *p = 15; // changing i via pointer
    cout << "After *p = 15, i = " << i << ", Address of i: " << &i << endl;

    printSeparator("Copying a Pointer");

    int *q = p; // q is another pointer to i
    cout << "Address of q: " << &q << ", q points to address: " << q << endl;
    *q = 20;
    cout << "After *q = 20, i = " << i << ", Address of i: " << &i << endl;

    printSeparator("Pointer to Pointer");

    int **r = &p;
    cout << "Value via **r: " << **r << endl;
    cout << "Address of r: " << &r << ", r points to address: " << r << ", **r = " << **r << endl;

    printSeparator("Static vs Dynamic Allocation");

    int a[10]; // static array on stack
    a[0] = 42;
    cout << "Static array a[0] = " << a[0] << ", Address of a: " << &a << ", Address of a[0]: " << &a[0] << endl;

    int size = 5;
    int *dynArray = new int[size]; // dynamic allocation on heap
    for (int j = 0; j < size; ++j)
        dynArray[j] = j * 10;
    cout << "Dynamic array contents: ";
    for (int j = 0; j < size; ++j)
        cout << dynArray[j] << " ";
    cout << endl;
    cout << "Address of dynArray: " << &dynArray << ", Address of dynArray[0]: " << &dynArray[0] << endl;

    delete[] dynArray;  // free memory
    dynArray = nullptr; // avoid dangling pointer

    printSeparator("Pointer Safety Check");

    if (dynArray != nullptr)
        *dynArray = 100;
    if (dynArray)
        *dynArray = 200; // same as above

    printSeparator("References");

    int k = 7;
    int &ref = k; // reference to k
    cout << "k = " << k << ", ref = " << ref << ", Address of k: " << &k << ", Address of ref: " << &ref << endl;

    ref = 13; // change k via ref
    cout << "After ref = 13, k = " << k << ", Address of k: " << &k << endl;

    // int& bad;         // ❌ not allowed: must initialize a reference
    // ref = another_var; // ❌ ref can't be reassigned

    return 0; // return 0 is standard success exit code
}
```

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

C++ at a glance

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

void printSeparator(const string &label)
{
    cout << "\n=== " << label << " ===\n";
}

int main()
{
    printSeparator("Basic Pointer Example");

    int i = 7;
    int *p = &i; // p points to i
    cout << "Value of i: " << i << ", Address of i: " << &i << ", *p: " << *p << endl;
    cout << "Value of i: " << i << ", Address of i: " << &i << ", *p: " << *p << ", Address of p: " << &p << endl;
    *p = 15; // changing i via pointer
    cout << "After *p = 15, i = " << i << ", Address of i: " << &i << endl;

    printSeparator("Copying a Pointer");

    int *q = p; // q is another pointer to i
    cout << "Address of q: " << &q << ", q points to address: " << q << endl;
    *q = 20;
    cout << "After *q = 20, i = " << i << ", Address of i: " << &i << endl;

    printSeparator("Pointer to Pointer");

    int **r = &p;
    cout << "Value via **r: " << **r << endl;
    cout << "Address of r: " << &r << ", r points to address: " << r << ", *r = " << *r << endl;

    printSeparator("Static vs Dynamic Allocation");

    int a[10]; // static array on stack
    a[0] = 42;
    cout << "Static array a[0] = " << a[0] << ", Address of a: " << &a << ", Address of a[0]: " << &a[0] << endl;

    int size = 5;
    int *dynArray = new int[size]; // dynamic allocation on heap
    for (int j = 0; j < size; ++j)
        dynArray[j] = j * 10;
    cout << "Dynamic array contents: ";
    for (int j = 0; j < size; ++j)
        cout << dynArray[j] << " ";
    cout << endl;
    cout << "Address of dynArray: " << &dynArray << ", Address of dynArray[0]: " << &dynArray[0] << endl;

    delete[] dynArray;  // free memory
    dynArray = nullptr; // avoid dangling pointer

    printSeparator("Pointer Safety Check");

    if (dynArray != nullptr)
        *dynArray = 100;
    if (dynArray)
        *dynArray = 200; // same as above

    printSeparator("References");

    int k = 7;
    int &ref = k; // reference to k
    cout << "k = " << k << ", ref = " << ref << ", Address of k: " << &k << ", Address of ref: " << &ref << endl;

    ref = 13; // change k via ref
    cout << "After ref = 13, k = " << k << ", Address of k: " << &k << endl;

    // int& bad;        // ❌ not allowed: must initialize a reference
    // ref = another_var; // ❌ ref can't be reassigned

    return 0; // return 0 is standard success exit code
}
```

```
~/Desktop/kr-uw/amath83-spr2025/lecture-codes% ./xptr-ref

=== Basic Pointer Example ===
Value of i: 7, Address of i: 0x16d0ab510, *p: 7
After *p = 15, i = 15

=== Copying a Pointer ===
After *q = 20, i = 20

=== Pointer to Pointer ===
Value via **r: 20

=== Static vs Dynamic Allocation ===
Static array a[0] = 42
Dynamic array contents: 0 10 20 30 40

=== Pointer Safety Check ===

=== References ===
k = 7, ref = 7
After ref = 13, k = 13
~/Desktop/kr-uw/amath83-spr2025/lecture-codes% g++ -std=c++17 -c ptr-ref.cpp; g++ -std=c++17 -o xptr-ref ptr-ref.o
~/Desktop/kr-uw/amath83-spr2025/lecture-codes% ./xptr-ref

=== Basic Pointer Example ===
Value of i: 7, Address of i: 0x16d693510, *p: 7
Value of i: 7, Address of i: 0x16d693510, *p: 7, Address of p: 0x16d693508
After *p = 15, i = 15, Address of i: 0x16d693510

=== Copying a Pointer ===
Address of q: 0x16d6934e8, q points to address: 0x16d693510
After *q = 20, i = 20, Address of i: 0x16d693510

=== Pointer to Pointer ===
Value via **r: 20
Address of r: 0x16d6934c8, r points to address: 0x16d693508, **r = 20

=== Static vs Dynamic Allocation ===
Static array a[0] = 42, Address of a: 0x16d693540, Address of a[0]: 0x16d693540
Dynamic array contents: 0 10 20 30 40
Address of dynArray: 0x16d6934a0, Address of dynArray[0]: 0x6000002a5200

=== Pointer Safety Check ===

=== References ===
k = 7, ref = 7, Address of k: 0x16d693464, Address of ref: 0x16d693464
After ref = 13, k = 13, Address of k: 0x16d693464
```

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

End Lecture 6

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025