04. A Tour of C++: Memory

Data Structure and Algorithms

Hanjun Kim

Compiler **O**ptimization **Re**search **Lab**

Yonsei University

Last time: Flow

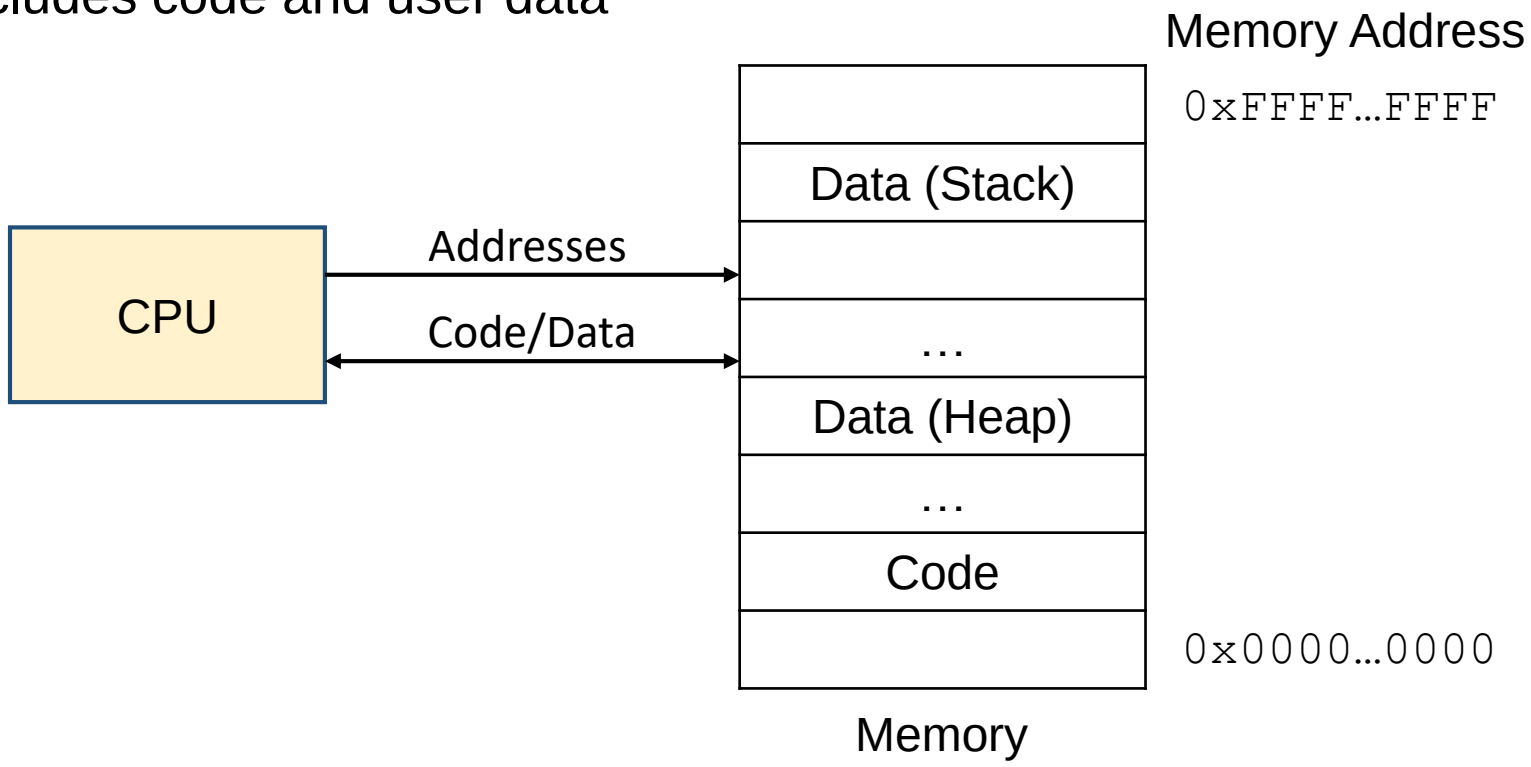
- Conditionals
  - IF statement
  - Switch-case statement
- Loops
  - While loop
  - Do While loop
  - For loop
- Nesting conditionals and loops
- Error Handling
  - Error
  - Assertion
  - Exception

Today: Memory

- Pointers
- References
- Pointers and Arrays
  - Arrays
  - Pointer Arithmetics
- Dynamic Allocation
- Pointers and const
- Function Pointers
- Memory Layout

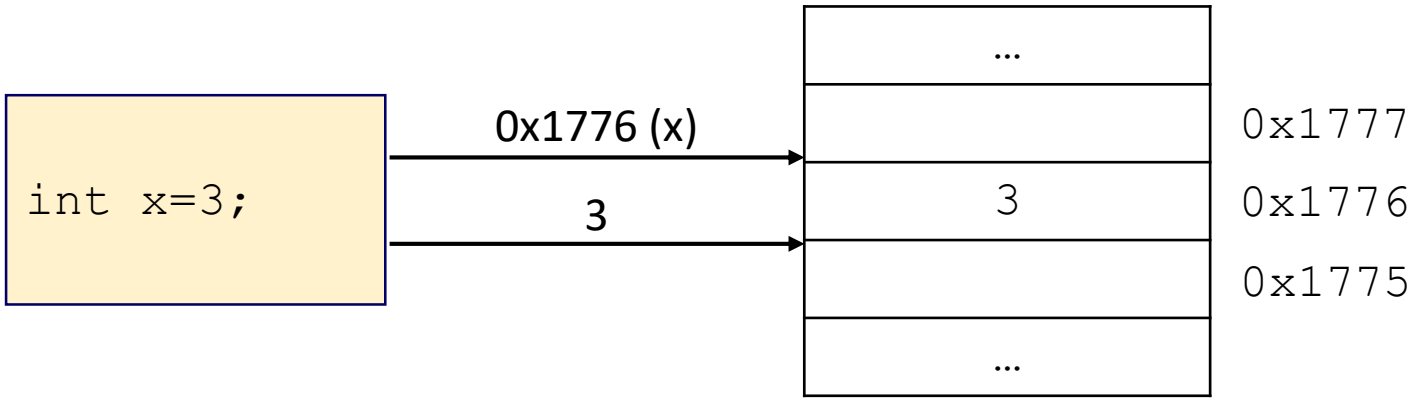
A Computer Memory

- Memory
  - Byte addressable array
  - Consists of addresses and contents
  - Includes code and user data



Variable in A Computer Memory View

- Variable
  - Memory location that can be accessed by its identifier
  - Generally, an OS decides the memory location on runtime
  - Sometimes, obtaining the memory address is useful for a program to access data (It is why we need pointers)



Pointers

- Pointer
  - A variable that stores the address of another variable
  - Declaration

```
type * name;
```

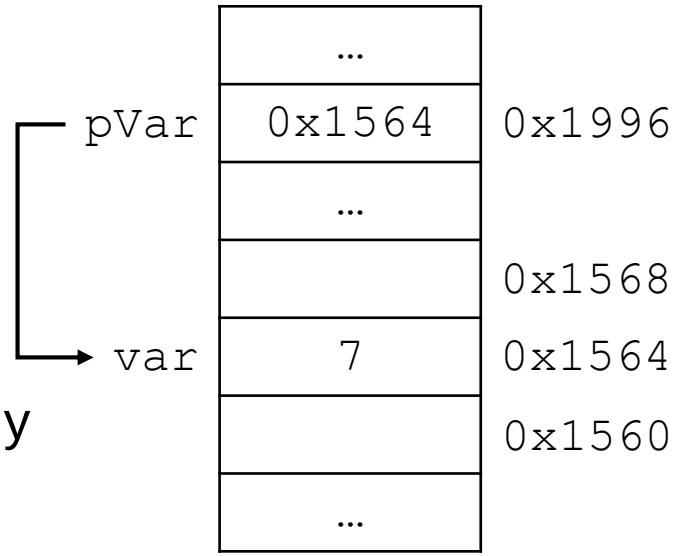
    - `type` is the data type pointed to by the pointer

- Address-of operator (&)
  - Returns the address of a variable

```
int var = 7;
int *pVar = &var;
```

- Dereference operator (*)
  - Access the variable that it points to directly

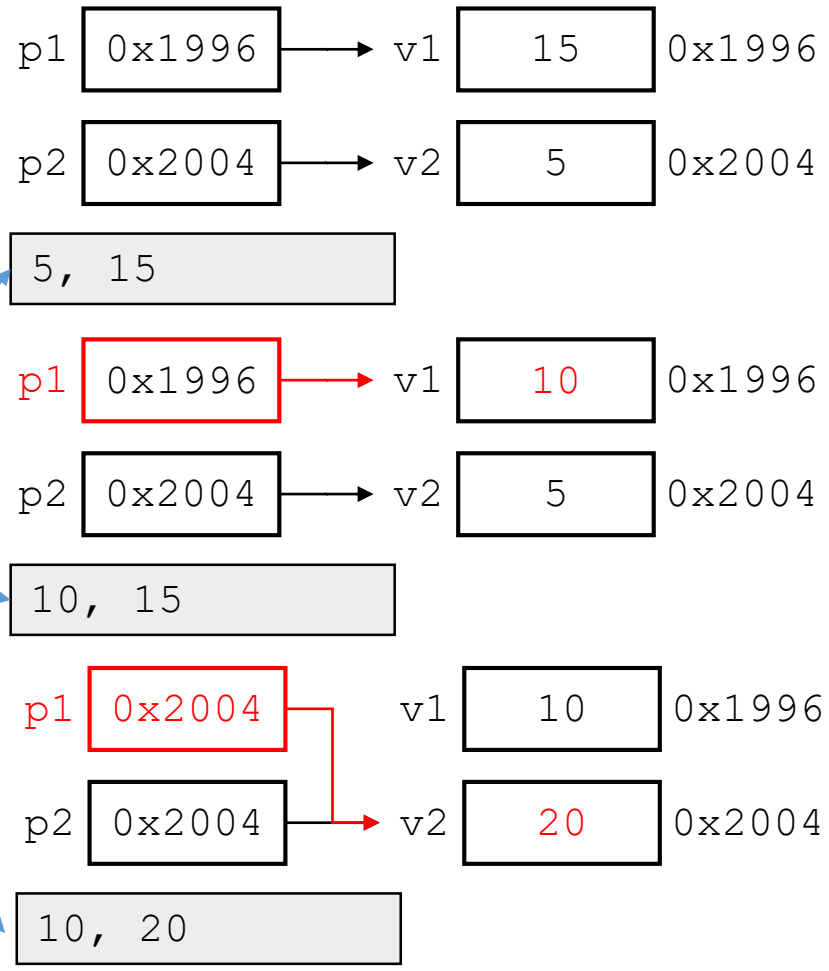
```
cout << pVar;     \\ 0x1564
cout << *pVar;    \\ 7
cout << &pVar;    \\ 0x1996
```



Pointers: Example

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

int main (){
  int v1 = 5, v2 = 15;
  int *p1, *p2;
  p1 = &v1; p2 = &v2;
  cout << v1 << ", " << v2 << endl;


  *p1 = 10;
  cout << v1 << ", " << v2 << endl;


  p1 = p2;
  *p1 = 20;
  cout << v1 << ", " << v2 << endl;
  return 0;
}
```

pointer.cpp

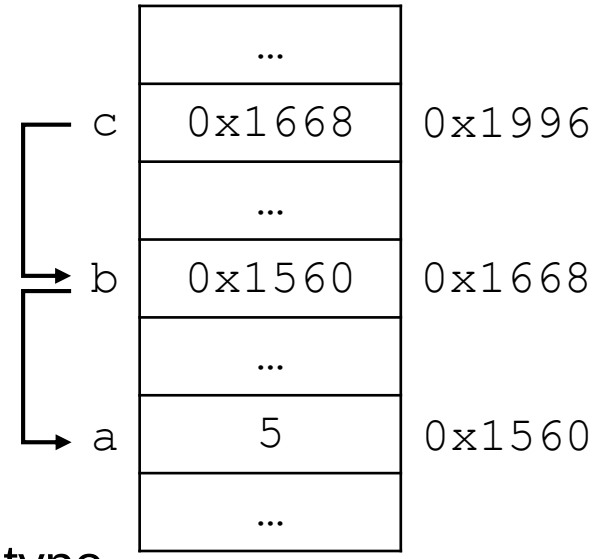


Pointers (cont')

- Double Pointer
  - A pointer can point to another pointer

```
int a = 5;
int* b = &a;
int** c = &b;
```

  - Here, `**c == 5`



- `void` Pointer
  - A pointer that points to a value that has no type

```
int iV = 5;
char cV = 'a';
void *p1;
```

```
p1 = &iV;    \\ valid
p1 = &cV;    \\ valid
```

- Null Pointer
  - A pointer that explicitly points to nowhere

```
int * p = 0;
int * q = nullptr;
```

Pointers: Void Pointer Example

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

void increase (void* data, int pSize) {
  if(pSize == sizeof(char)) {
    char* pChar;
    pChar=(char*)data;
    ++(*pChar);
  } else if(pSize == sizeof(int)){
    int* pInt;
    pInt=(int*)data;
    ++(*pInt);
  }
}

int main() {
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

void_pointer.cpp

```
y, 1603
```

Reference

- Reference (Reference type, Reference variable)
  - An alternative name for a memory object (Alias)
  - Declaration

```
type & name;
```

    - `type` is the data type referenced by name
  - Example

```
int A = 5;
int& rA = A;
```

  - References cannot be null nor uninitialized

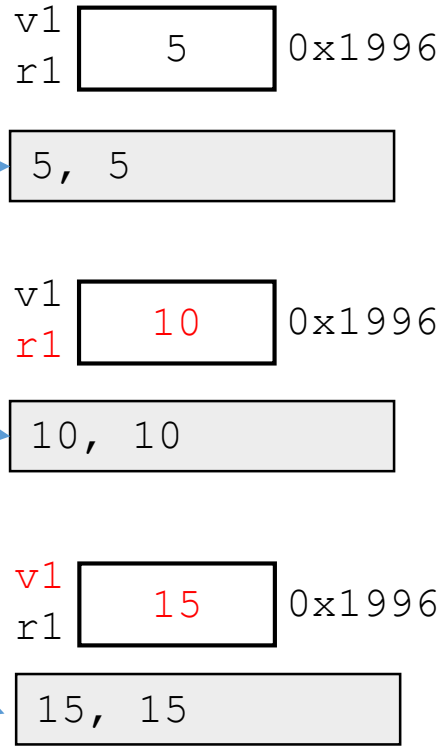
```
int& k; // compile error:
        // k declared as reference but not initialized
```

  - Once a reference is created, it cannot be later made to reference another object
  - Different from address-of operator (&)

Reference Example

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

int main (){
  int v1 = 5;
  int &r1 = v1;
  cout << v1 << ", " << r1 << endl;



  r1 = 10;
  cout << v1 << ", " << r1 << endl;



  v1 = 15;
  cout << v1 << ", " << r1 << endl;
  return 0;
}
```

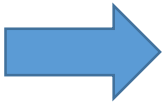
reference.cpp



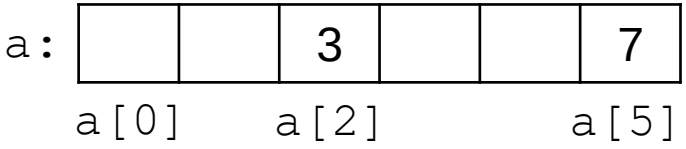
Arrays

- Array
  - Indexed sequence of values of the same type
  - Examples
    - 52 playing cards in a deck
    - 18 thousand undergrads at Yonsei
    - 1 million characters in a book
    - 10 million audio samples in an MP3 file

```
int a0, a1, a2, a3, a4, a5;
a2 = 3;
a5 = 7;
int x = a2 + a5;
```



```
int a[6];
a[2] = 3;
a[5] = 7;
int x = a[2] + a[5];
```



Arrays

- Array declaration

```
type name [elements];
```

- Array initialization
  - Elements in an array can be explicitly initialized to specific values when it is declared by enclosing those initial values in braces {}
  - Example

```
int foo [5] = { 6, 2, 7, 4, 9};
```

foo:

6	2	7	4	9

- Multidimensional arrays
  - Arrays of arrays

```
int foo [3][2] = { {6, 2}, {7, 4}, {9, 3}};
```

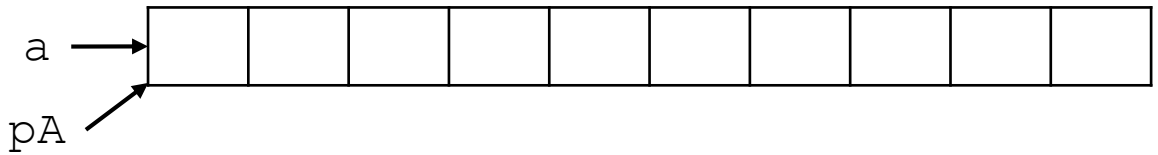
Pointers and Arrays

- Arrays work like pointers to their first elements
  - `A` and `&A[0]` are the same for an array `A`
  - `[ ]` (brackets)
    - Index of an element of the array
    - Dereferencing operator known as offset operator

```
a[5] = 0;        // a [offset of 5] = 0
```

- An array can always be converted to a pointer
  - Example

```
int a[10];
int *pA;
pA = a;
```



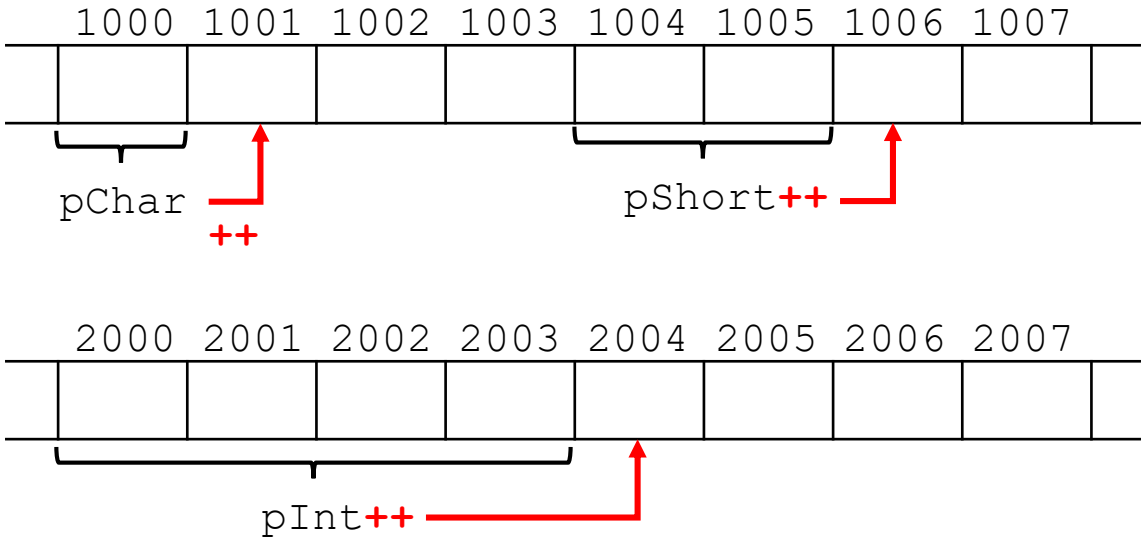
- However, a pointer cannot be converted to an array

```
a = pA;  // illegal
```

Pointer Arithmetics

- Pointer Arithmetics
  - Arithmetical operations on pointers
  - Only addition and subtraction operations are allowed
  - Show different behavior according to the size of its data type
    - +1 means the next element in an array
  - Example (char: 1 byte, short: 2 bytes, int: 4 bytes)

```
char *pChar;
short *pShort;
int *pInt;

pChar++;
pShort;
pInt++;
```



Pointers and Arrays Example

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

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;              // numbers[0] = 10
  p++;  *p = 20;                      // numbers[1] = 20
  p = &numbers[2];  *p = 30;          // numbers[2] = 30
  p = numbers + 3;  *p = 40;          // numbers[3] = 40
  p = numbers;  *(p+4) = 50;          // numbers[4] = 50
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

array.cpp

```
10, 20, 30, 40, 50,
```

Dynamic Allocation

- Motivation
  - All memory needs to be determined before program execution
  - Memory needs of a program may be determined during runtime
- Dynamic allocation
  - `new` Operator: dynamically allocates memory

```
pointer = new type
```

```
pointer = new type [number_of_elements]
```

  - `delete` Operator: deallocates previously allocated memory

```
delete pointer
```

```
delete[] pointer
```

  - Example

```
int * foo = new int [5];
…
delete[] foo;
```

Dynamic Allocation

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

int main (){
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr) exit(-1);

  for (n=0; n<i; n++){
    cout << "Enter number: ";
    cin >> p[n];
  }
  cout << "You have entered: ";
  for (n=0; n<i; n++)
    cout << p[n] << ", ";
  delete[] p;
  return 0;
}
```

dynamic_allocation.cpp

Pointers and const

- const
  - A notion of immutability
  - Declares a pointer that will not change its pointed value

```
int x;
      int *       p1 = &x;  // non-const pointer to non-const int
const int *       p2 = &x;  // non-const pointer to const int
int const *       p3 = &x;  // non-const pointer to const int
      int * const p4 = &x;  // const pointer to non-const int
const int * const p5 = &x;  // const pointer to const int
```

  - Example

```
int x;
int y = 10;
const int * p = &y;
x = *p;           // ok: reading p
*p = x;           // error: modifying p, which is const-qualified
```

Function Pointers

- Function Pointer
  - A pointer that points to a function (A function is also located in memory)
  - Typical usage: Passing a function as an argument to another function

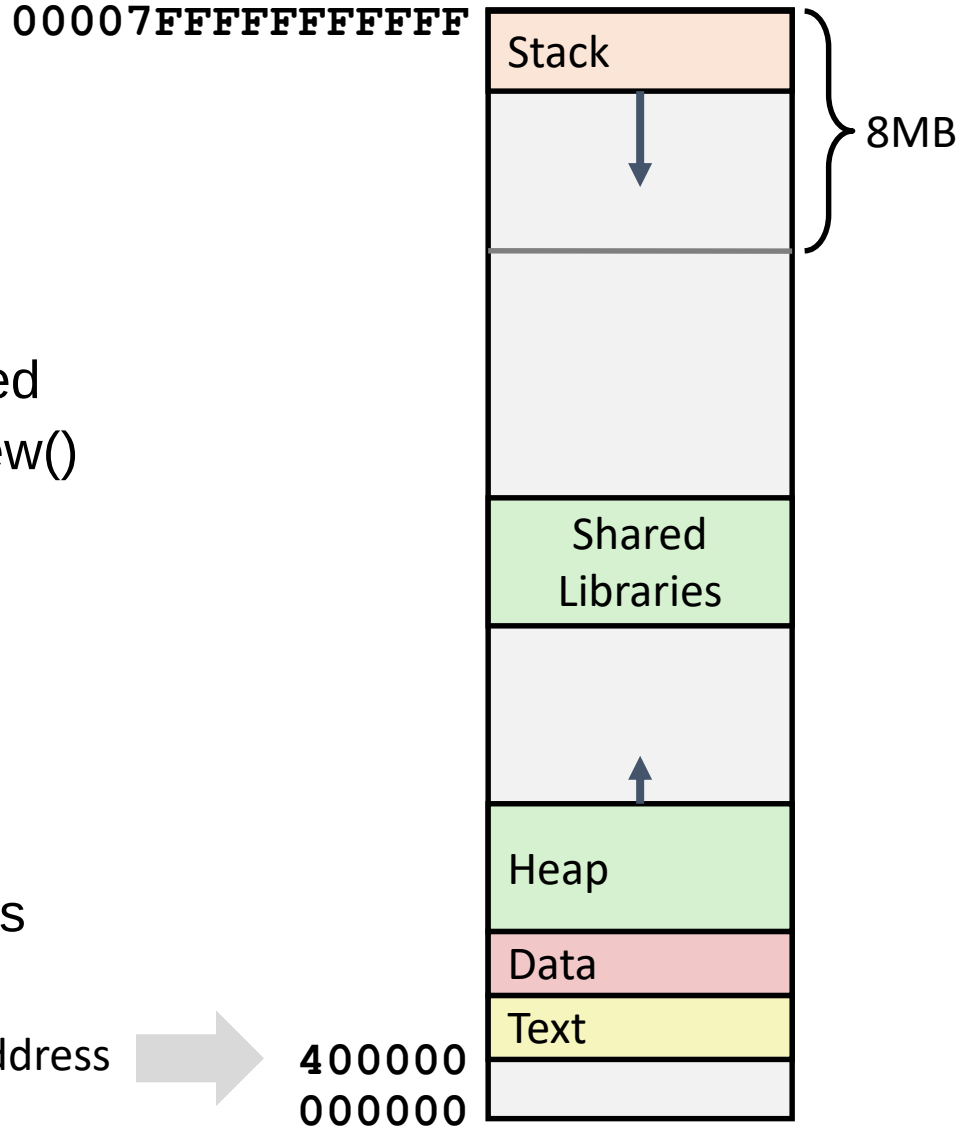
```
#include <iostream>
using namespace std;

int addition(int a, int b){ return (a+b); }
int subtraction(int a, int b){ return (a-b); }

int operation (int x, int y, int (*pF)(int,int)){
  int g = (*pF)(x,y);
  return g;
}

int main (){
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

functionPointer.cpp

```
8
```

x86-64 Linux Memory Layout

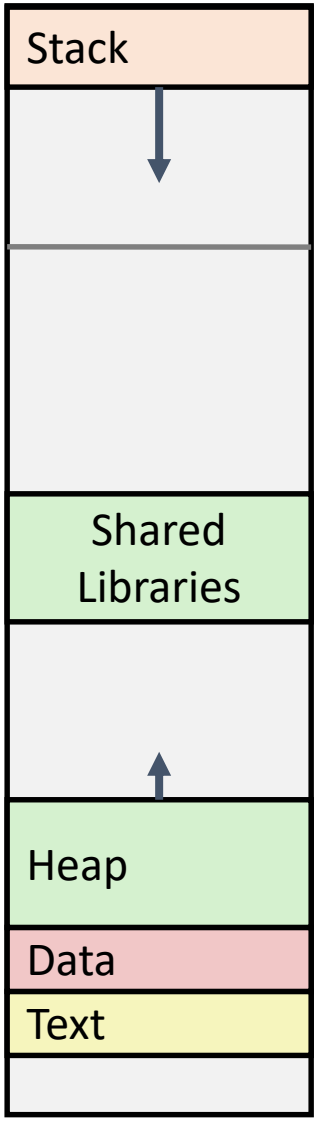
- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

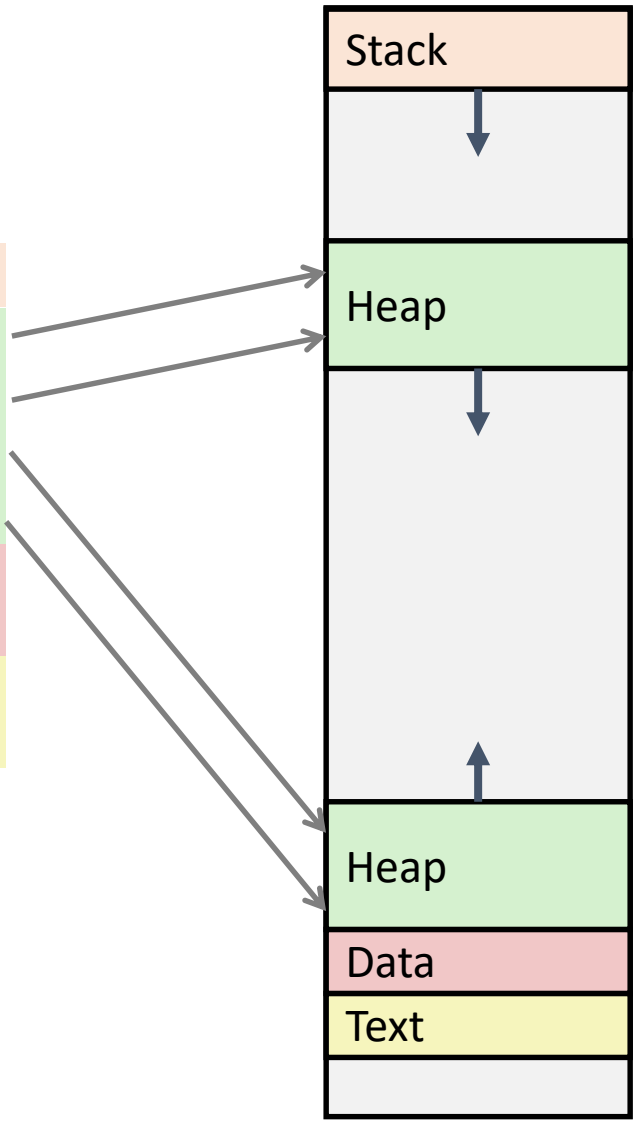
Where does everything go?



x86-64 Example Addresses

address range ~2^{47}

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



Summary: Memory

- Pointers
  - Address-of (&) and Dereference (*) Operators
  - Double, void and null pointers
- References
- Pointers and Arrays
  - Arrays
  - Pointer Arithmetics
- Dynamic Allocation
- Pointers and const
- Function Pointers
- Memory Layout