Creative Software Programming

4 – Dynamic Memory Allocation, References

Yoonsang Lee
Fall 2019

Questions from Last Lecture

- **using namespace *ns*;**
  - "**Import**" the namespace *ns* into the **current scope**
  - Subsequent code will use identifiers in the namespace *ns* **as if they were in current namespace**
  - This effect applies only **within the scope "using namespace" used**

```
#include <iostream>
using namespace std; // import std into global
scope

namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}
int main () {
   using namespace first_space; // import
first_space into the current scope (main())

   // at this moment, std and first_space are
imported

   func(); // first_space::func();
   return 0;
}
```

Questions from Last Lecture

```
#include <iostream>
using namespace std; // import std into global
scope

namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}
int main () {
   using namespace first_space; // import
first_space into the current scope (main())

   func(); // first_space::func();

   using namespace second_space; // import
second_space into the current scope (main())

   // at this moment, std, first_space, and
second_space are imported
   func(); // so, generates an error

   return 0;
}
```

```
#include <iostream>
using namespace std; // import std into global
scope

namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}
int main () {
   {
      using namespace first_space; // import
first_space into the current scope
      func(); // first_space::func();
   }
   {
      using namespace second_space; // import
second_space into the current scope
      func(); // second_space::func();
   }
   return 0;
}
```

Questions from Last Lecture

- std::endl and '\n'
  - The same meaning: newline
  - The only difference is that std::endl **flushes** the output buffer, and '\n' doesn't.
    - flushing: transferring the data from the buffer to the stdout or file (and clear the buffer).

```
std::cout << std::endl;

// is equivalent to

std::cout << '\n' << std::flush;
```

Questions from Last Lecture

- Compound assignment operator (+=, -=, ...)
  - No compound assignment for logical operators in C++
  - But &= operator is available. I guess &&= is a typo.
  - https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Compound_assignment_operators

- Is "string2" geting deleted from the memory after str2 = "string22"?
  - String literal is in read-only memory segment, so no modification or removal is allowed at runtime.

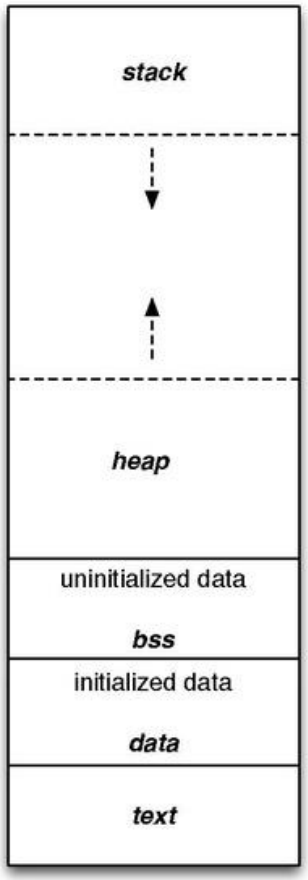
Today's Topics

- Dynamic Memory Allocation
  - Typical Memory Layout of C / C++ Programs
  - malloc() / free() and new / delete
  - Memory leak
- References
  - Differences btwn. Pointer & Reference
  - When to use Pointer / Reference?

Dynamic Memory Allocation

Typical Memory Layout of C / C++ Programs

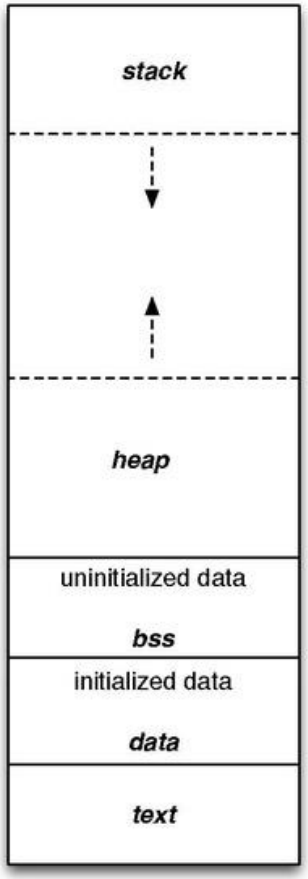
- When you run a C / C++ program, OS allocates memory space for the program like this:



Organized in several segments:

- Stack segment
- Heap segments
- BSS segments
- Data segments
- Text segments

Typical Memory Layout of C / C++ Programs



: local variables, function parameters, return address

: for dynamic memory allocation

(data segment)

: global variables, static variables

(code segment)

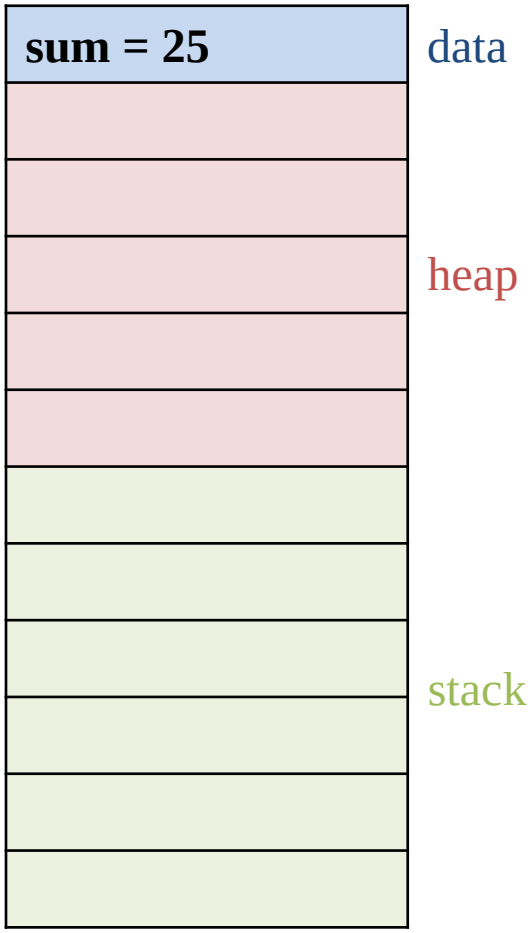
: compiled binary code of the program

- The reason of "typical" is, it's actually platform / implementation dependent (not a part of C/C++ specifications), but it generally used in most popular platforms.

Example - Memory Layout 1

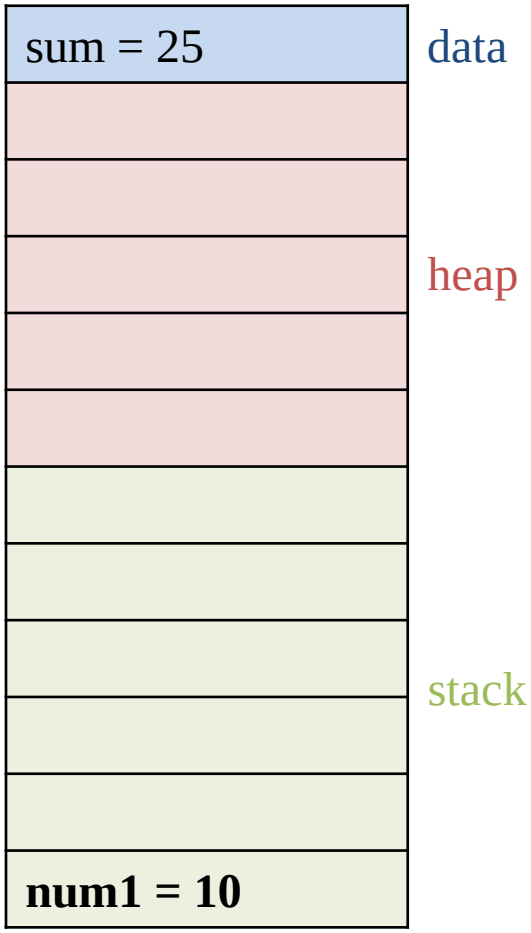
(Program starts)

```
int sum=25;

int main()
{
   int num1=10;
   func(num1);
   num++;
   func(num1);
   return 0;
}

void func(int n)
{
   int num2=20;
}
```



Example - Memory Layout 2

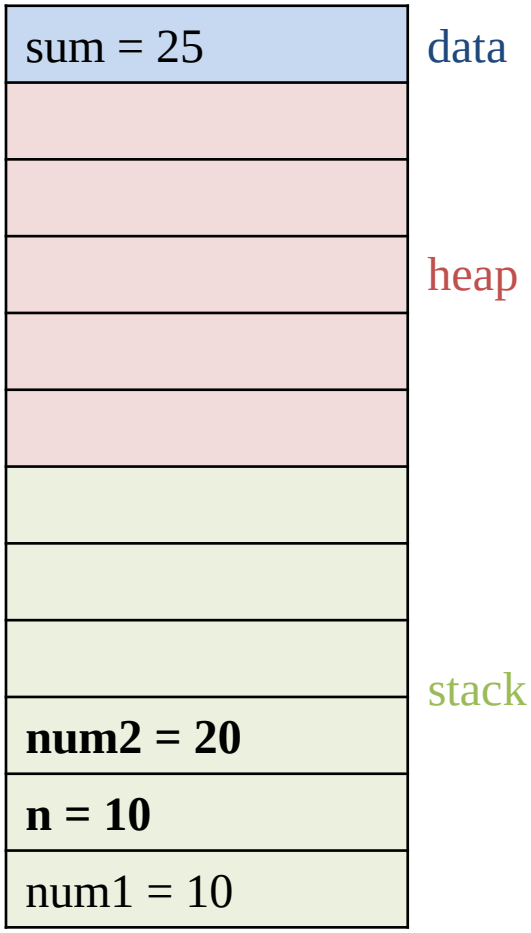
```
int sum=25;

int main()
{
   int num1=10;
   func(num1);
   num++;
   func(num1);
   return 0;
}

void func(int n)
{
   int num2=20;
}
```



Example - Memory Layout 3

```
int sum=25;

int main()
{
   int num1=10;
   func(num1);
   num++;
   func(num1);
   return 0;
}

void func(int n)
{
   int num2=20;
}
```

call



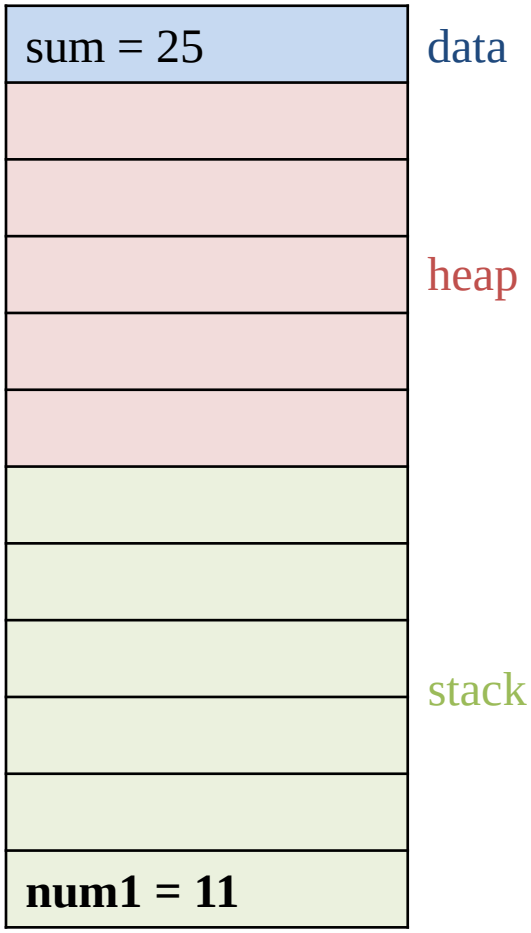
data

heap

stack

Example - Memory Layout 4

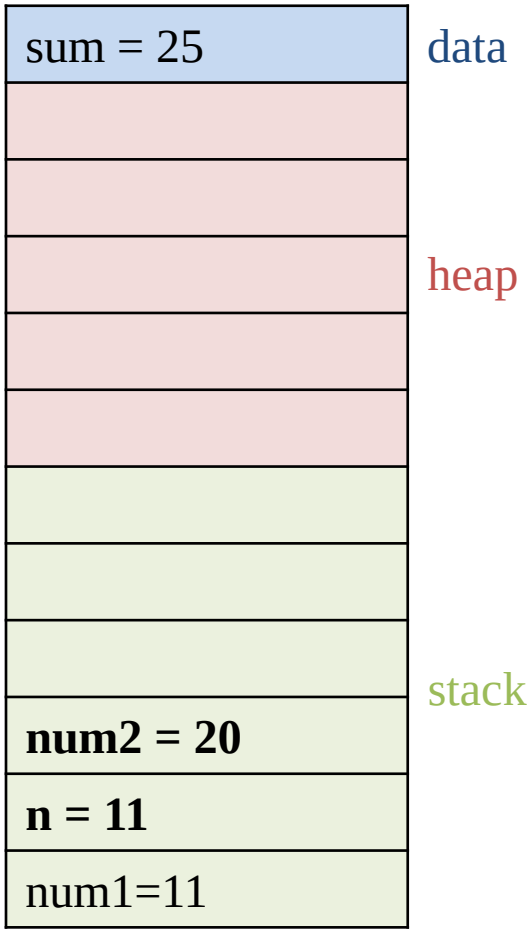
```
int sum=25;

int main()
{
   int num1=10;
   func(num1);
   num++;
   func(num1);
   return 0;
}

void func(int n)
{
   int num2=20;
}
```



Example - Memory Layout 5

```
int sum=25;

int main()
{
   int num1=10;
   func(num1);
   num++;
   func(num1);
   return 0;
}

void func(int n)
{
   int num2=20;
}
```

call



data

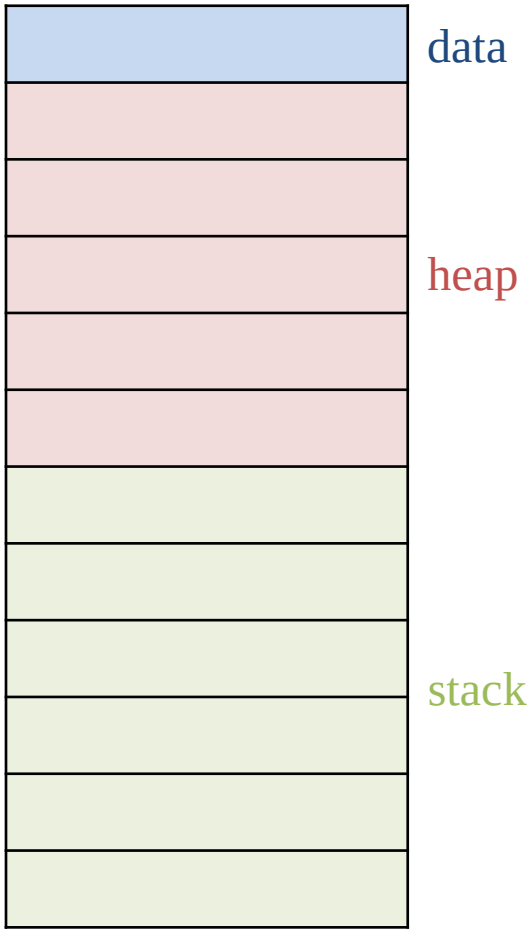
heap

stack

Example - Memory Layout 6

(Program ends)

```
int sum=25;

int main()
{
   int num1=10;
   func(num1);
   num++;
   func(num1);
   return 0;
}

void func(int n)
{
   int num2=20;
}
```



Dynamic Memory Allocation

- How to create an array whose length changes while the program is running?
- What if you could not determine the type and number of data to use when writing code?
- -> Your program has to **dynamically** allocate the necessary memory space during execution.
- Dynamically allocated data is store in the **heap**.

C malloc / free

- Allocate and deallocate memory block.
  - Example: C arrays are with fixed sizes.
  - How can we use variable size array?

```
void TestFunction(int n) {
  int fixed_size_array[20];
  int variable_size_array[n];  // Compile error.

  for (int i = 0; i < n; ++i) {
    cout << fixed_size_array[i] << ", "  // SEGFAULT if n > 20.
         << variable_size_array[i];
  }
}
```

  - (FYI) C99 standard supports variable-length array, but it's not encouraged to use. (https://en.wikipedia.org/wiki/Variable-length_array)

C malloc / free

- Allocate and deallocate memory block.
  - Use `malloc/free` to manage memory allocation.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void TestFunction(int n) {
  int* variable_size_array = (int*) malloc(sizeof(int) * n);
  for (int i = 0; i < n; ++i) {
    cout << variable_size_array[i] << endl;
  }
  free(variable_size_array);
}

int main() {
    TestFunction(3);
    return 0;
}
```

  - `malloc(n)` : allocates n bytes of memory block and return the pointer to the block.
  - `free(ptr)` : deallocates the allocated memory block.

C malloc / free

- What happens if allocated blocks are not freed?
- Memory leak : an allocated but unused memory is not returned to OS.
  - Usually happens when the pointer to it gets lost.

```
#include <stdlib.h>

void TestFunction(int n) {
  double* another_array = (double*) malloc(sizeof(double) * n);

  for (int i = 0; i < n; ++i) {
    int* variable_size_array = (int*) malloc(sizeof(int) * n);
    cin >> another_array[i]
        >> variable_size_array[i];
    // free(variable_size_array);
  }
  another_array = (double*) malloc(sizeof(double) * n);
  free(another_array);
}
```

Dynamic Memory Allocation

- C: malloc(), free() functions
  - `#include <stdlib.h>`
  - `int* pNum = (int*)malloc(sizeof(int));`
  - `free(pNum);`

- C++: **new**, **delete** operators
  - `int* pNum = new int;`
  - `delete pNum;`
  - Use this way in C++ (especially for class objects)

C++ new / delete

- C++ has `new` and `delete` operators built-in.
  - `new` : creates a variable(instance) of the type(class).
  - `delete` : destructs a variable(instance) created by `new`.
  - `new []` : creates an **array** of variables(instances) of the type(clas s).
  - `delete[]` : destructs an **array** created by `new[]`.

	One instance	Array
Allocate	`new`	`new []`
Deallocate	`delete`	`delete[]`

Examples - Dynamic Memory Allocation 1

C version

```c
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
    int n;
    cin >> n;

    // allocate one instance
    int* num = (int*)malloc(sizeof(int));
    // allocate an array
    int* numArr = (int*)malloc(sizeof(int)*n);

    *num = n;
    for(int i=0; i<n; i++)
        numArr[i] = i;

    cout << *num << endl;
    for(int i=0; i<n; i++)
        cout << numArr[i] << " ";
    cout << endl;

    free(num);    // deallocate the instance
    free(numArr); // deallocate the array

    return 0;
}
```

C++ version

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

int main()
{
    int n;
    cin >> n;

    // allocate one instance
    int* num = new int;
    // allocate an array
    int* numArr = new int[n];

    *num = n;
    for(int i=0; i<n; i++)
        numArr[i] = i;

    cout << *num << endl;
    for(int i=0; i<n; i++)
        cout << numArr[i] << " ";
    cout << endl;

    delete num;      // deallocate the instance
    delete[] numArr; // deallocate the array

    return 0;
}
```

Examples - Dynamic Memory Allocation 2

C version

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void TestFunction(int n) {
  int* int_instance = (int*)
malloc(sizeof(int));
  int* variable_size_array = (int*)
malloc(sizeof(int) * n);

  *int_instance = 10;
  for (int i = 0; i < n; ++i)
    cin >> variable_size_array[i];

  free(int_instance);
  free(variable_size_array);
}

int main() {
    TestFunction(3);
    return 0;
}
```

C++ version

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void TestFunction(int n) {
  int* int_instance = new int;
  int* variable_size_array = new int[n];

  *int_instance = 10;
  for (int i = 0; i < n; ++i)
    cin >> variable_size_array[i];

  delete int_instance;
  delete[] variable_size_array;
}

int main() {
    TestFunction(3);
    return 0;
}
```

Quiz #1

- Go to https://www.slido.com/
- Join #csp-hyu
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked as "attendance".

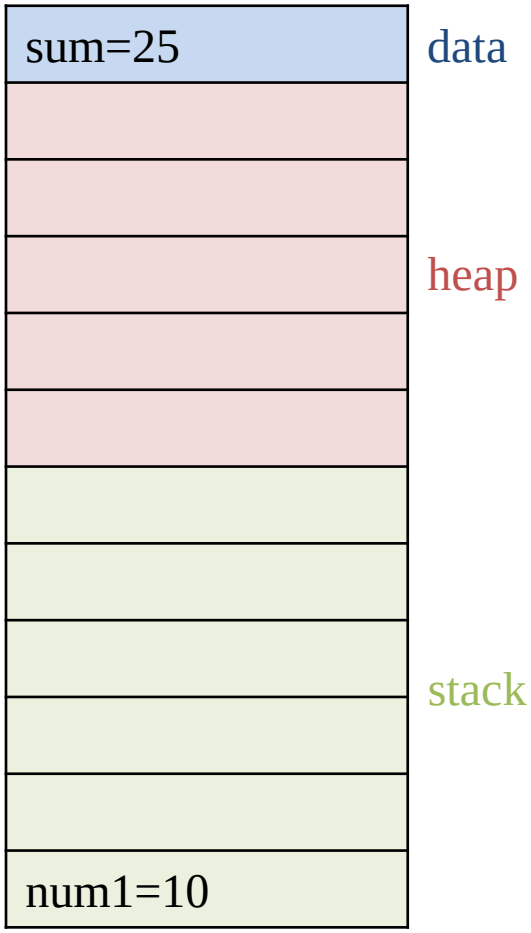
C++ new / delete

- Just like C malloc() / free(), C++ new / delete can cause memory leak.

- Be sure to call delete every time you call new.
  - Always use new and delete in pairs.
  - Do not call new and delete in different functions (More likely to make a mistake not to call delete).

Example - Memory Layout (Dynamic Alloc.) 1

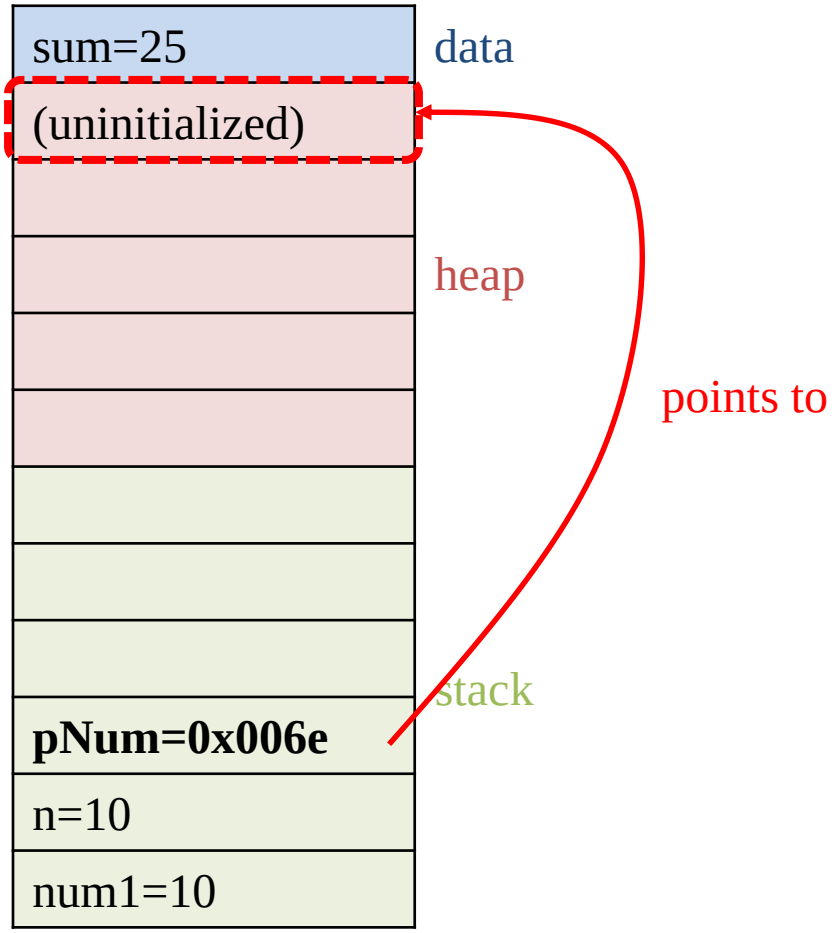
```
int sum=25;

int main(void)
{
  int num1=10;
  fct(num1);
  num++;
  fct(num1);
  return 0;
}

void fct(int n)
{
  int* pNum = new int;
  *pNum = n;
  delete pNum;
}
```



Example - Memory Layout (Dynamic Alloc.) 2

```
int sum=25;

int main(void)
{
   int num1=10;
   fct(num1);
   num++;
   fct(num1);
   return 0;
}

void fct(int n)
{
   int* pNum = new int;
   *pNum = n;
   delete pNum;
}
```

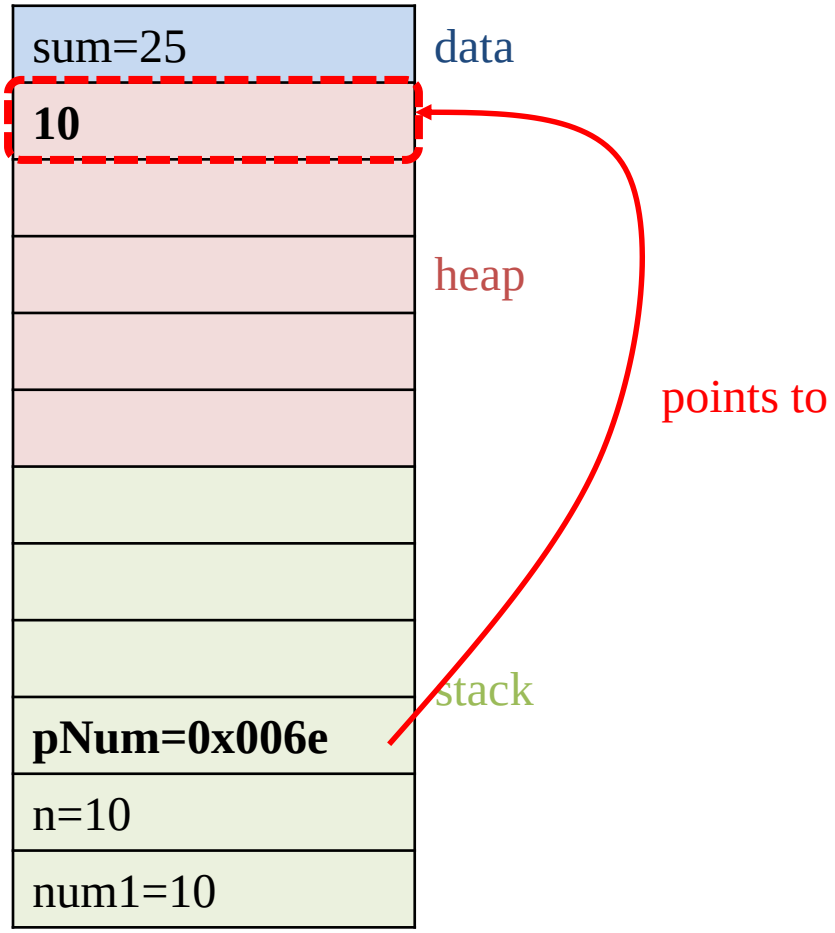
call



Example - Memory Layout (Dynamic Alloc.) 3

```
int sum=25;

int main(void)
{
   int num1=10;
   fct(num1);
   num++;
   fct(num1);
   return 0;
}

void fct(int n)
{
   int* pNum = new int;
   *pNum = n;
   delete pNum;
}
```

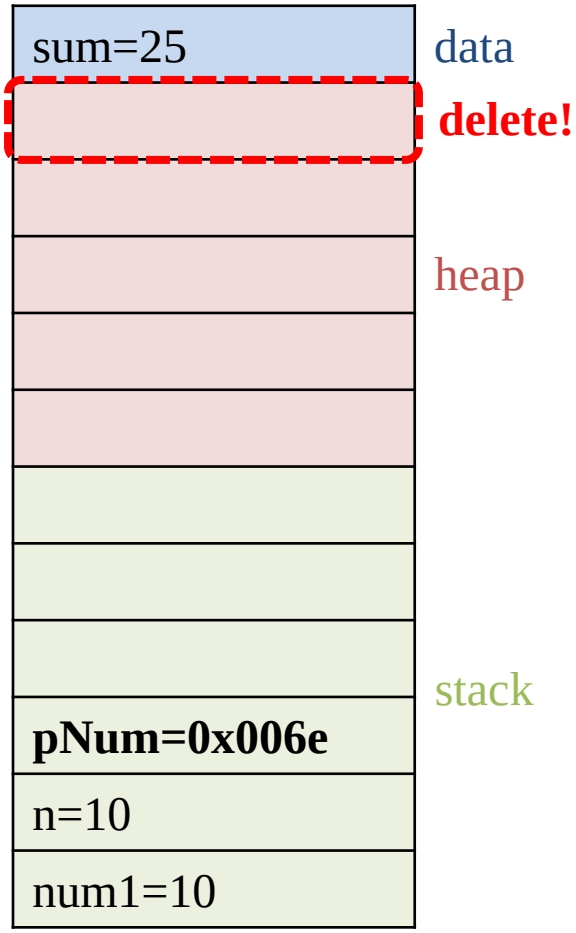
call



Example - Memory Layout (Dynamic Alloc.) 4

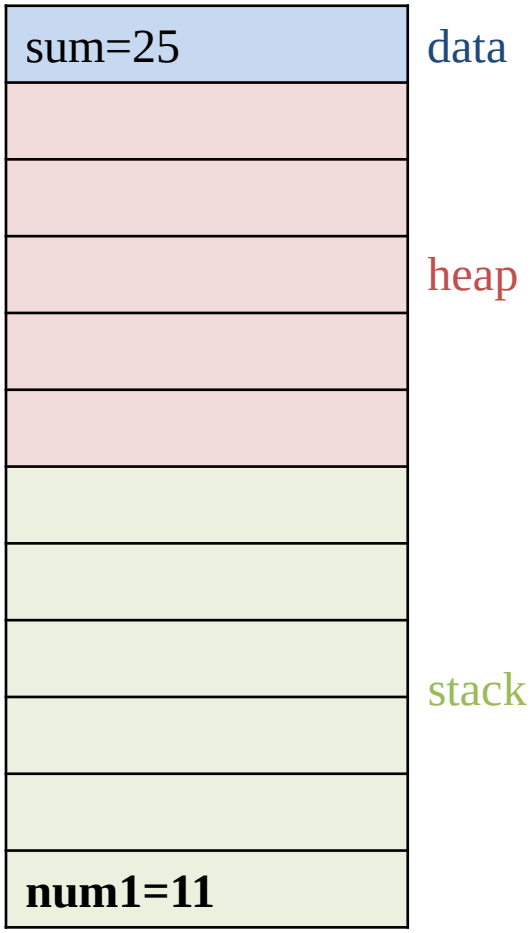
```
int sum=25;

int main(void)
{
   int num1=10;
   fct(num1);
   num++;
   fct(num1);
   return 0;
}

void fct(int n)
{
   int* pNum = new int;
   *pNum = n;
   delete pNum;
}
```

call



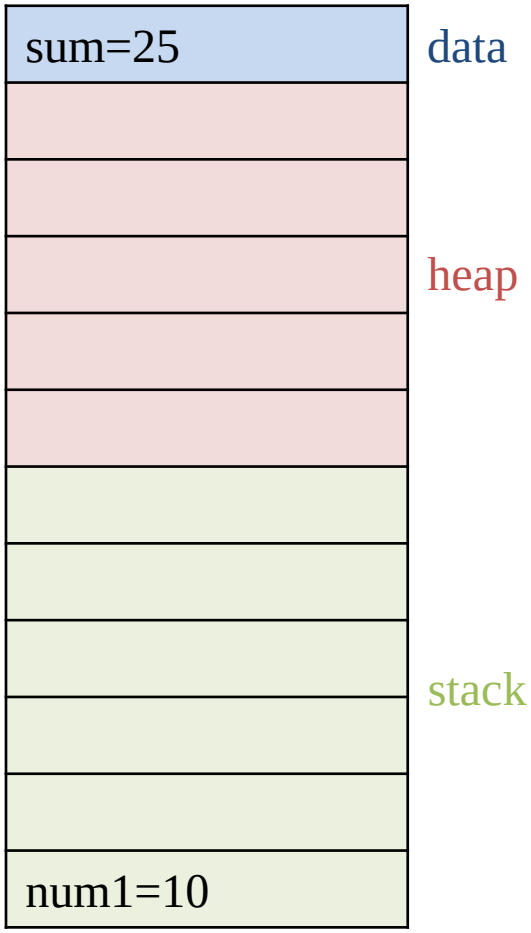
Example - Memory Layout (Dynamic Alloc.) 5

```
int sum=25;

int main(void)
{
  int num1=10;
  fct(num1);
  num++;
  fct(num1);
  return 0;
}

void fct(int n)
{
  int* pNum = new int;
  *pNum = n;
  delete pNum;
}
```



Example - Memory Layout (Memory Leak) 1

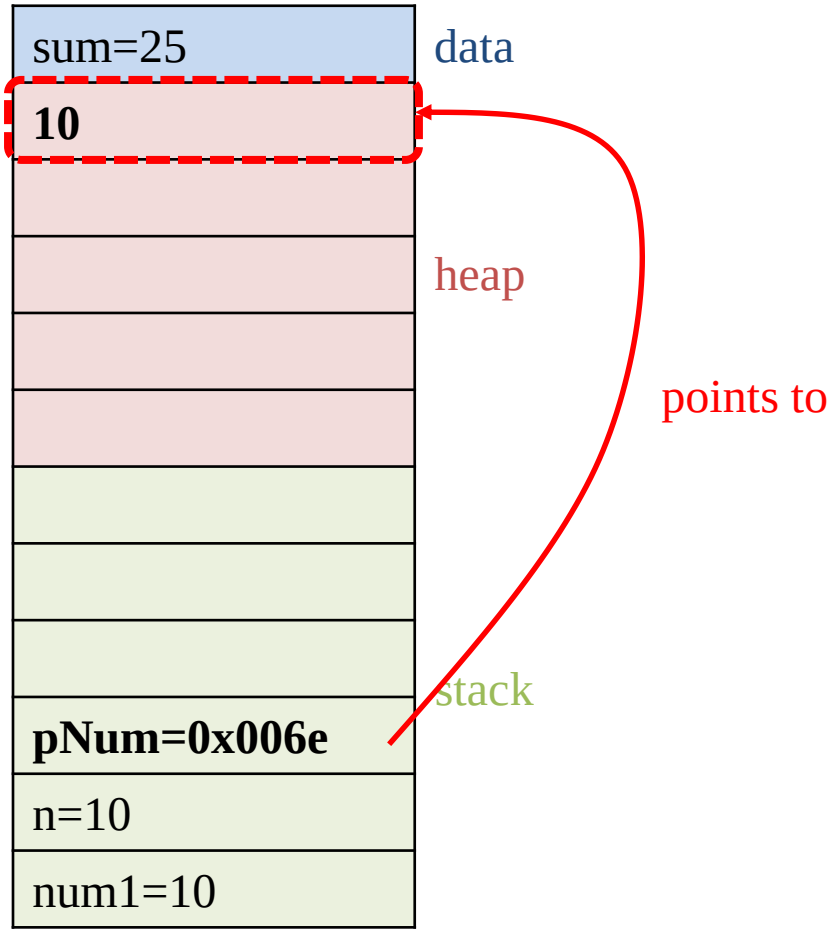
```
int sum=25;

int main(void)
{
  int num1=10;
  fct(num1);
  num++;
  fct(num1);
  return 0;
}

void fct(int n)
{
  int* pNum = new int;
  *pNum = n;
  //delete pNum;
}
```



Example - Memory Layout (Memory Leak) 2

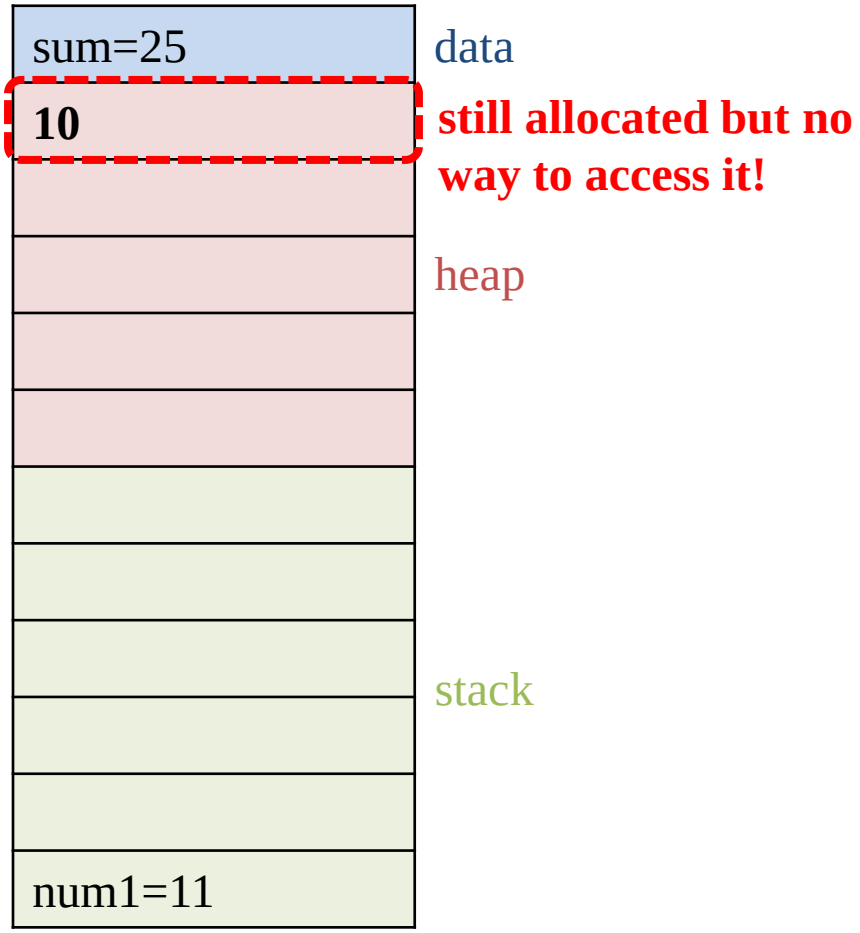
```
int sum=25;

int main(void)
{
   int num1=10;
   fct(num1);
   num++;
   fct(num1);
   return 0;
}

void fct(int n)
{
   int* pNum = new int;
   *pNum = n;
   //delete pNum;
}
```

call



Example - Memory Layout (Memory Leak) 3

```
int sum=25;

int main(void)
{
   int num1=10;
   fct(num1);
   num++;
   fct(num1);
   return 0;
}

void fct(int n)
{
   int* pNum = new int;
   *pNum = n;
   //delete pNum;
}
```



Quiz #2

- Go to https://www.slido.com/
- Join #csp-hyu
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked as "attendance".

References

C++ Reference (&)

- References can be used similar to pointers (Think of it as a "referenced pointer")
  - Less powerful but safer than the pointer type.

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

int main()
{
    int a = 10;
    int* pa = &a; // pa can be regarded as an "alias" of a
    *pa = 20;
    cout << a << " " << *pa << endl;    // 20 20


    int b = 10;
    int& rb = b; // rb can be regarded as an "alias" of b
    rb = 20;
    cout << b << " " << rb << endl;    // 20 20

    return 0;
}
```

Differences btwn. Pointer & Reference

- A pointer can be uninitialized

```
int* pa;  // ok
```

- A reference must be initialized

```
int& rb;      // error

int b = 10;
int& rb = b;  // ok
```

Differences btwn. Pointer & Reference

- A pointer can be reassigned

```
int a=1, b=2;
int* p;
p = &a;
p = &b;
```

- A reference cannot be reassigned (must be initialized)

```
int a=1, b=2;
int& r = a;
r = b; // Not referencing b, just copy value of b to a

r = 100;
cout << a << " " << b << " " << r << endl; // 100 2 100
```

Differences btwn. Pointer & Reference

- A pointer can point to a null object (NULL or nullptr in c++11)

```
int* p = NULL;   // ok
```

- A reference cannot refer to a null object

```
int& r = NULL;   // error
```

Recall: When to use Pointers in C?

- Passing read-only parameters to a function
  - Recall: `void printPoint(const Point* p)`
  - C/C++ parameter passing and returning **copy the data**
  - If a function does not need to modify the value of passed variables, use **"pointer to constant"** to **avoid copying**

- You can use **references** for this purpose as well!
  - `void printPoint(const Point& p)`

Passing by Reference to Constant

- Passing arguments using const reference type (const &)
  - The instances **remains unchanged after the function call.**
  - Avoids copying the arguments.
  - Guarantees reference to a valid instance.

```
struct Triplet { int a, b, c; };

void TestConstReference(const Triplet ct, const Triplet* cpt,
                        const Triplet& crt) {
  ct.a = 10, cpt->b = 20, crt.c = 30;  // All are errors.
  printf("%d, %d, %d\n", ct.a, cpt->b, crt.c);
}

int main() {
  Triplet triplet;
  triplet.a = 10, triplet.b = 20, triplet.c = 30;

  TestConstReference(triplet, &triplet, triplet);
  return 0;
}
```

Recall: When to use Pointers in C?

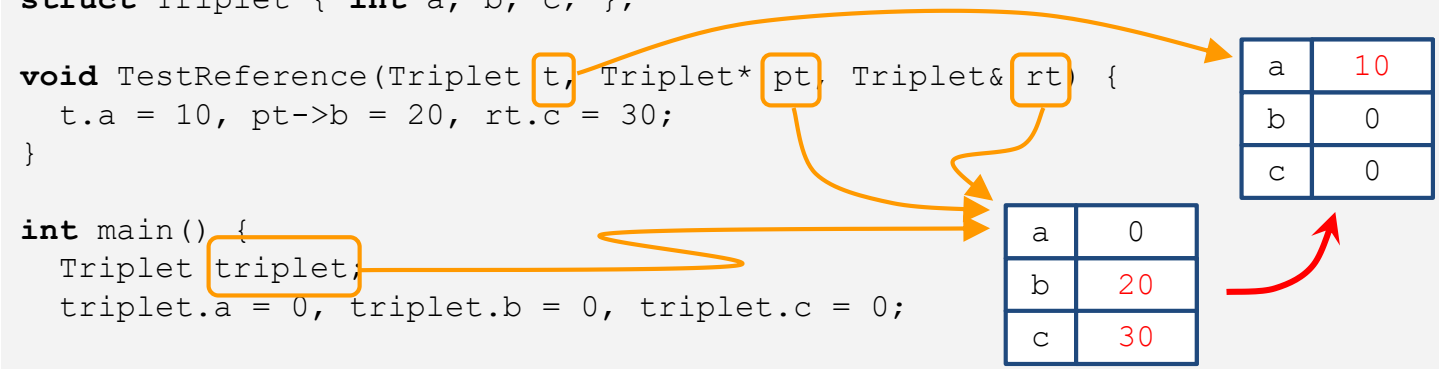
- Call-by-reference
  - Recall: `void swap(int* p1, int* p2)`
  - `swap` function can **modify** the value of passed variables
  - These parameters are often called ***out parameters***

- You can use **references** for this purpose as well!
  - `void swap(int& i1, int& i2)`

Passing by Reference

- Passing arguments using reference type (&)
  - The instances **probably are modified by the function.**
  - Avoids copying the arguments.
  - Guarantees reference to a valid instance (whereas pointer can be n ull)

```
struct Triplet { int a, b, c; };

void TestReference(Triplet t, Triplet* pt, Triplet& rt) {
  t.a = 10, pt->b = 20, rt.c = 30;
}

int main() {
  Triplet triplet;
  triplet.a = 0, triplet.b = 0, triplet.c = 0;

  TestReference(triplet, &triplet, triplet);
  // triplet.a == 0, triplet.b == 20, triplet.c == 30

  TestReference(triplet, NULL, triplet);  // Causes SEGFAULT.
  return 0;
}
```



Recall: When to use Pointers in C?

- Dynamic memory allocation
  - One has to use pointers to access memory on the **heap**
  - `int* pNum = (int*)malloc(sizeof(int));`
  - `int* pNum = new int;`

- References cannot be used for this purpose.

Quiz #3

- Go to https://www.slido.com/
- Join #csp-hyu
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked as "attendance".

DO NOT Confuse Address-of Operator(&) and Reference(&)!

- Address-of operator

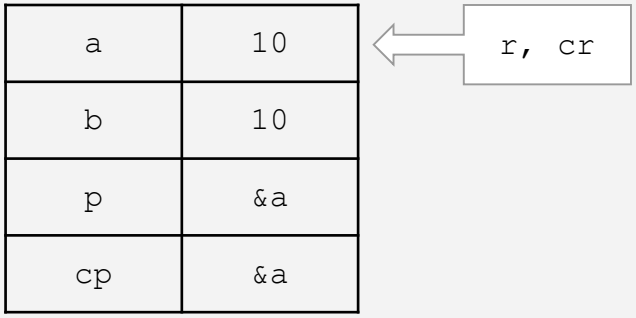
```
int a = 0;
int* pa = &a; // '&'+[variable name]
```

- Reference

```
int a = 0;
int& a_ref = a; // [type name]+'&'
```

Local Variable, Pointer, Reference

```
int a = 10;
int b = a;

int* p = &a;
const int* cp = &a;

int& r = a;
const int& cr = a;

a = 20;     // a: 20, b: 10, p: &a, *p: 20, cp: &a, *cp: 20, r: 20 ,cr: 20.
b = 30;     // a: 20, b: 30, p: &a, *p: 20, cp: &a, *cp: 20, r: 20 ,cr: 20.

*p = 10;    // a: 10, b: 30, p: &a, *p: 10, cp: &a, *cp: 10, r: 10 ,cr: 10.
*cp = 0;    // Error!
r = 40;     // a: 40, b: 30, p: &a, *p: 40, cp: &a, *cp: 40, r: 40 ,cr: 40.
cr = 0;     // Error!

p = &b;     // a: 40, b: 30, p: &b, *p: 30, cp: &a, *cp: 40, r: 40 ,cr: 40.
*p = 50;    // a: 40, b: 50, p: &b, *p: 50, cp: &a, *cp: 40, r: 40 ,cr: 40.

int** pp = &p;
*pp = &a;   // pp: &p, p: &a, *p: 40
*pp = &b;   // pp: &p, p: &b, *p: 50
```



Next Time

- Labs in this week:
  - Lab1: Assignment 4-1
  - Lab2: Assignment 4-2

- Next lecture:
  - 5 - Compilation and Linkage, CMD Args