Creative Software Design

Pointers and References in C++

Yunho Kim
yunhokim@hanyang.ac.kr
Dept. of Computer Science

Today's Topics

- Dynamic Memory Allocation
  - Typical Memory Layout of C / C++ Programs
  - malloc() / free() and new / delete
  - Memory leak

- References
  - What is the Reference?
  - Differences between Pointer & Reference
  - When to use Pointer / Reference?

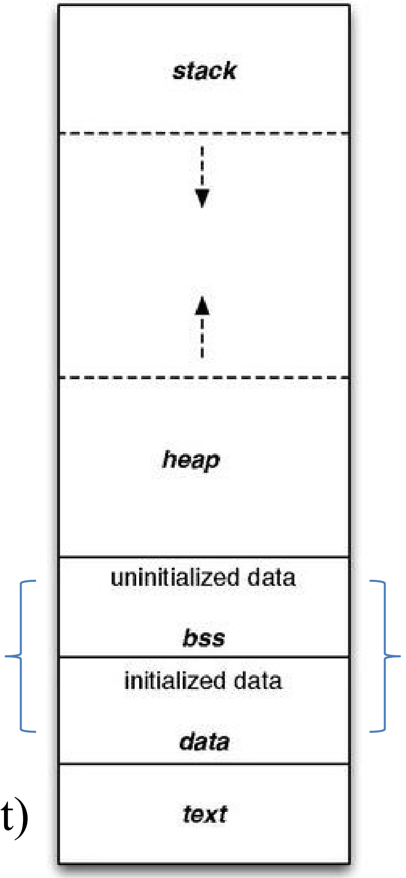
Dynamic Memory Allocation

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)

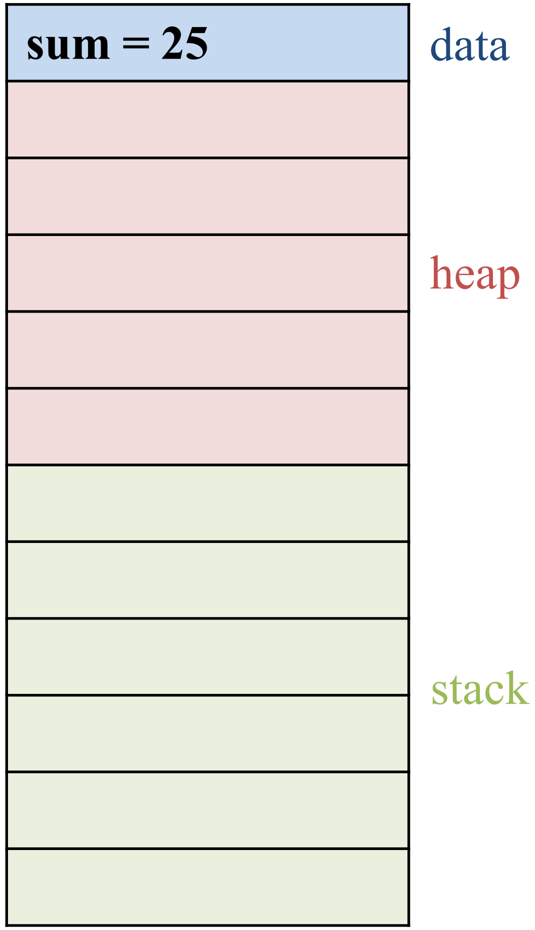
: compiled binary code of the program

- 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

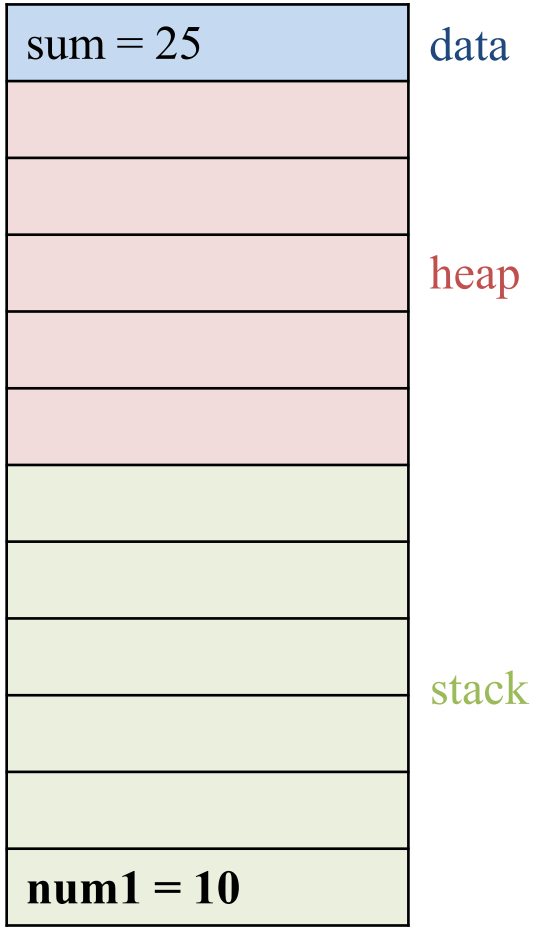
(Program starts)

```
int sum=25;

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

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



sum = 25

data

heap

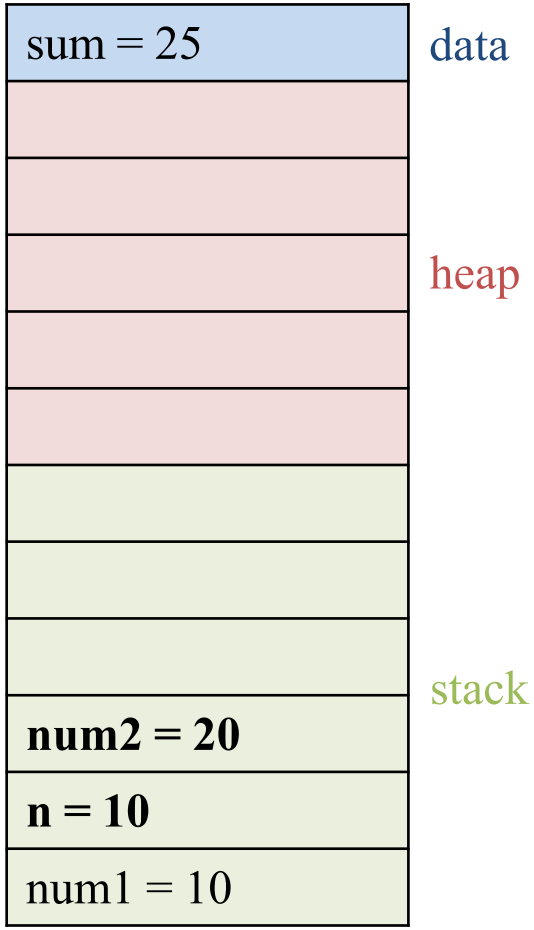
stack

num1 = 10

Example - Memory Layout 3

```
int sum=25;

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

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

call



sum = 25

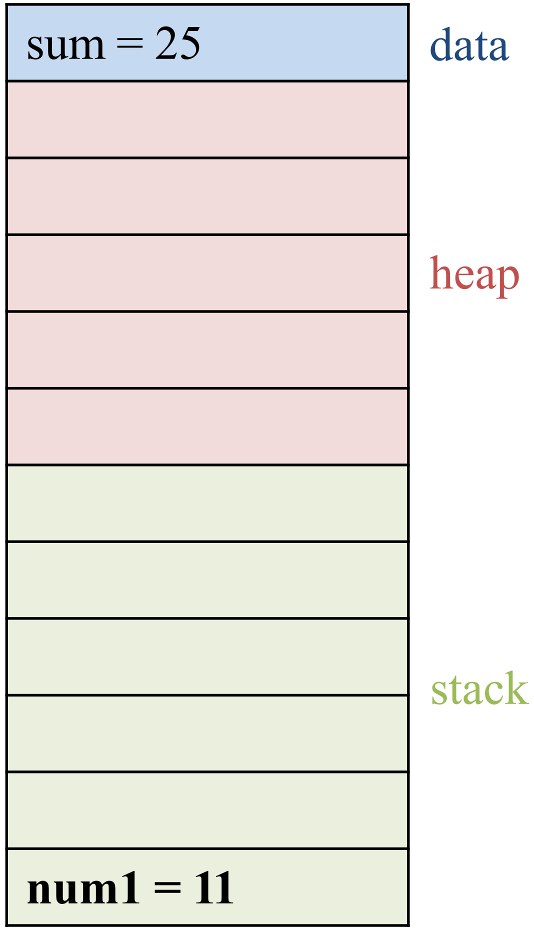
num2 = 20

n = 10

num1 = 10

Example - Memory Layout 4

```
int sum=25;

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

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



sum = 25

data

heap

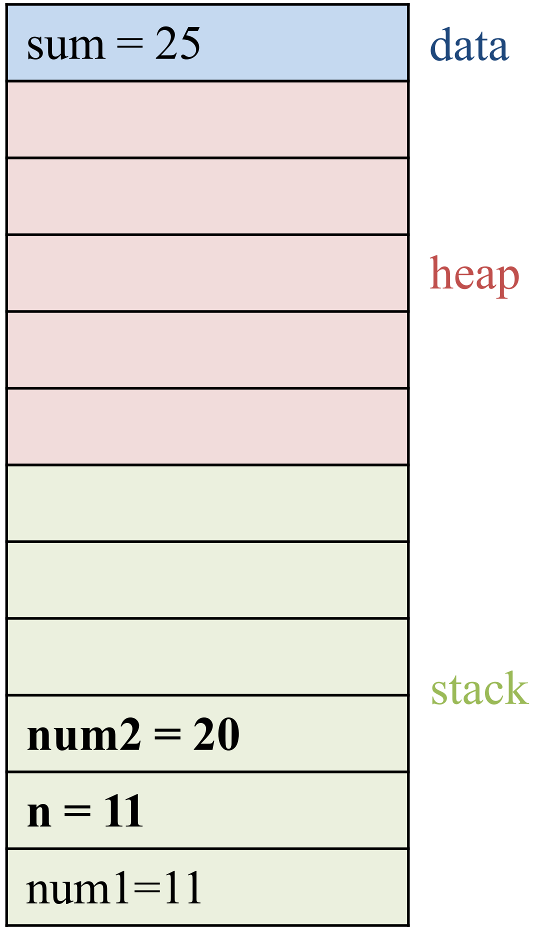
stack

num1 = 11

Example - Memory Layout 5

```
int sum=25;

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

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

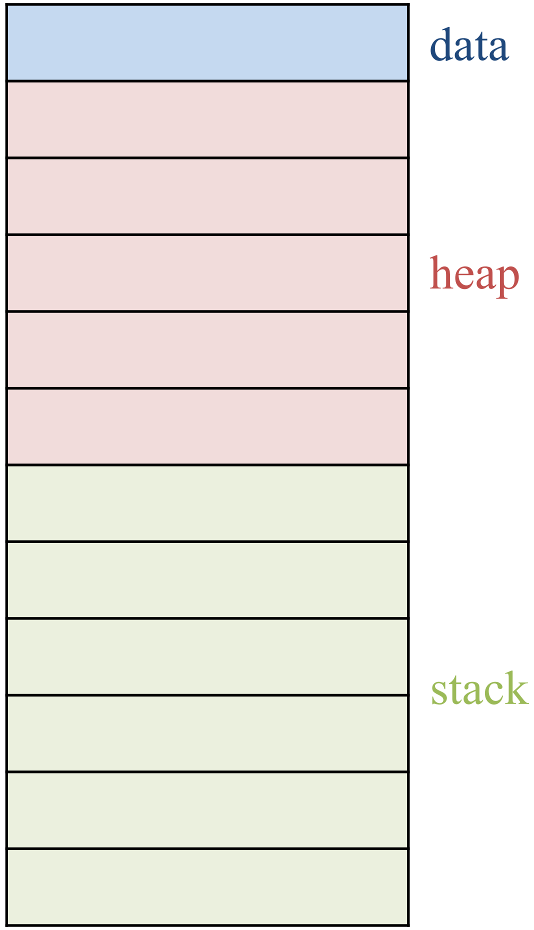
call



Example - Memory Layout 6

(Program ends)

```
int sum=25;

int main()
{
  int num1=10;
  func(num1);
  num1++;
  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**.

An Example

- 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 before C99.

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

  - (FYI) C99 and later standard supports a variable-length array (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 <cstdlib>
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;
}
```

In C, (int *) is optional
In C++, (int *) is required because implicit type casting from (void *) to another pointer type is not allowed

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

Dynamic Memory Allocation

- C: malloc(), free() functions
  - `#include <cstdlib>`
  - `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(class).
  - `delete[]` : destructs an **array** created by `new[]`.

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

Examples - Dynamic Memory Allocation 1

C-style

```
#include <iostream>
#include <cstlib>
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++ style

```
#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-style

```
#include <iostream>
#include <cstlib>
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++-style

```
#include  <iostream>
#include    <cstlib>
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;
}
```

Memory Leak

- What happens if allocated blocks are not freed?

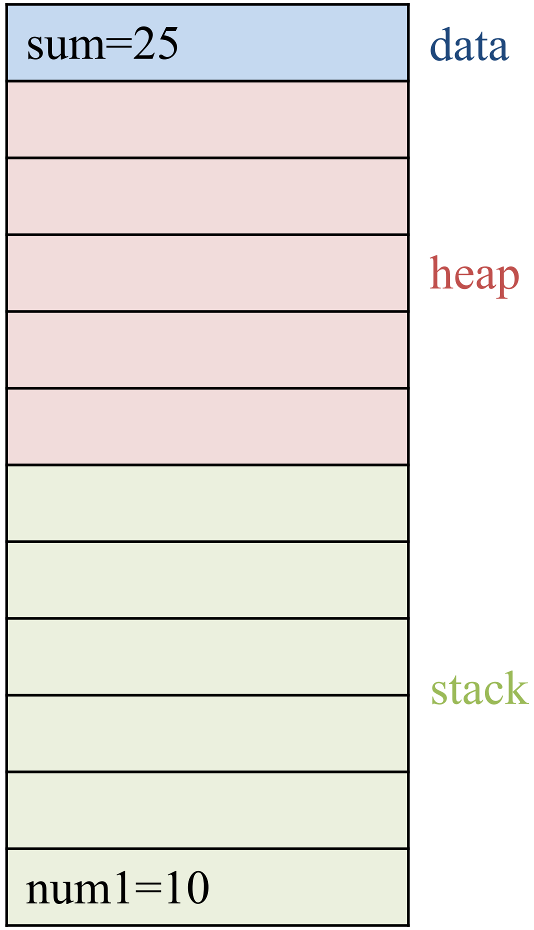
- Memory leak : an allocated but unused memory is not returned to OS.
  - Usually happens when the pointer to it gets lost.

- 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

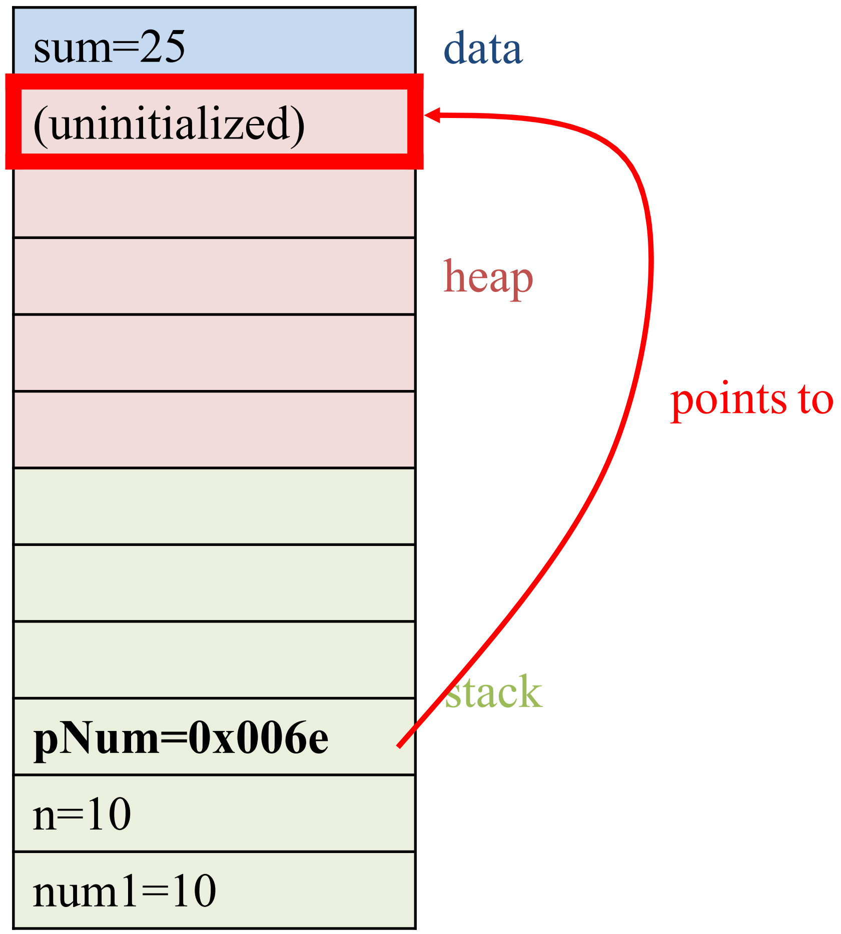
```
int sum=25;

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

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



Example - Memory Layout (Dynamic Alloc.) 2

call

```
int sum=25;

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

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



sum=25

data

(uninitialized)

heap

points to

stack

pNum=0x006e

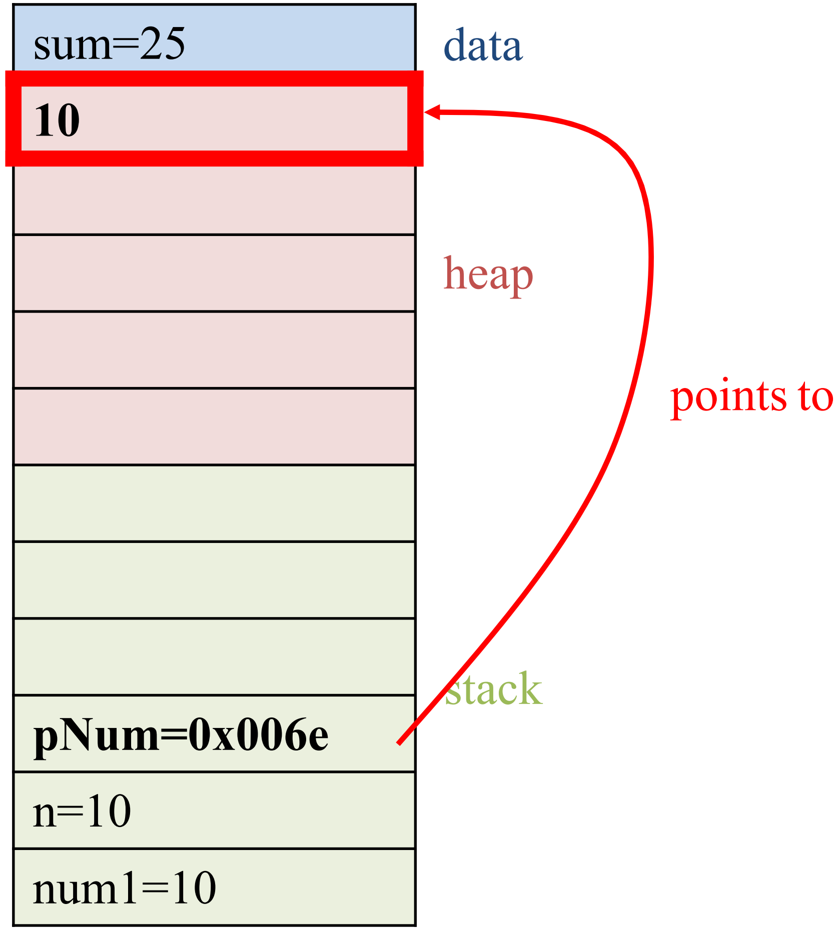
n=10

num1=10

Example - Memory Layout (Dynamic Alloc.) 3

call

```
int sum=25;

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

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



sum=25

data

10

heap

points to

stack

pNum=0x006e

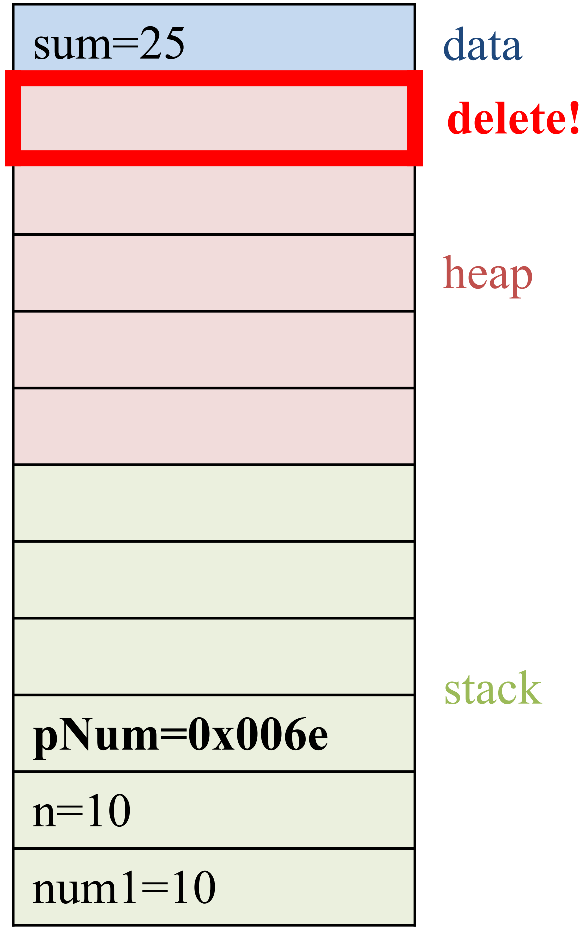
n=10

num1=10

Example - Memory Layout (Dynamic Alloc.) 4

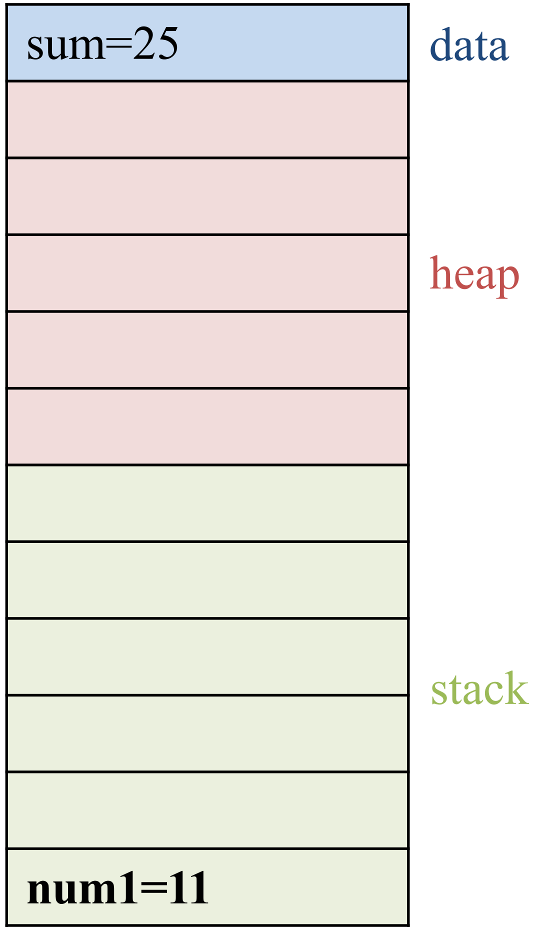
call

```
int sum=25;

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

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



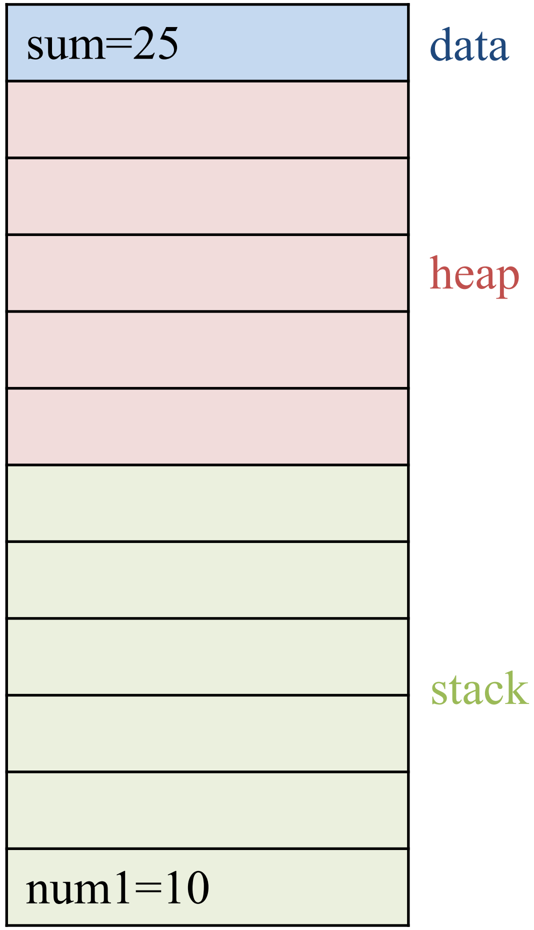
Example - Memory Layout (Dynamic Alloc.) 5

```
int sum=25;

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

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



Example - Memory Layout (Memory Leak) 1

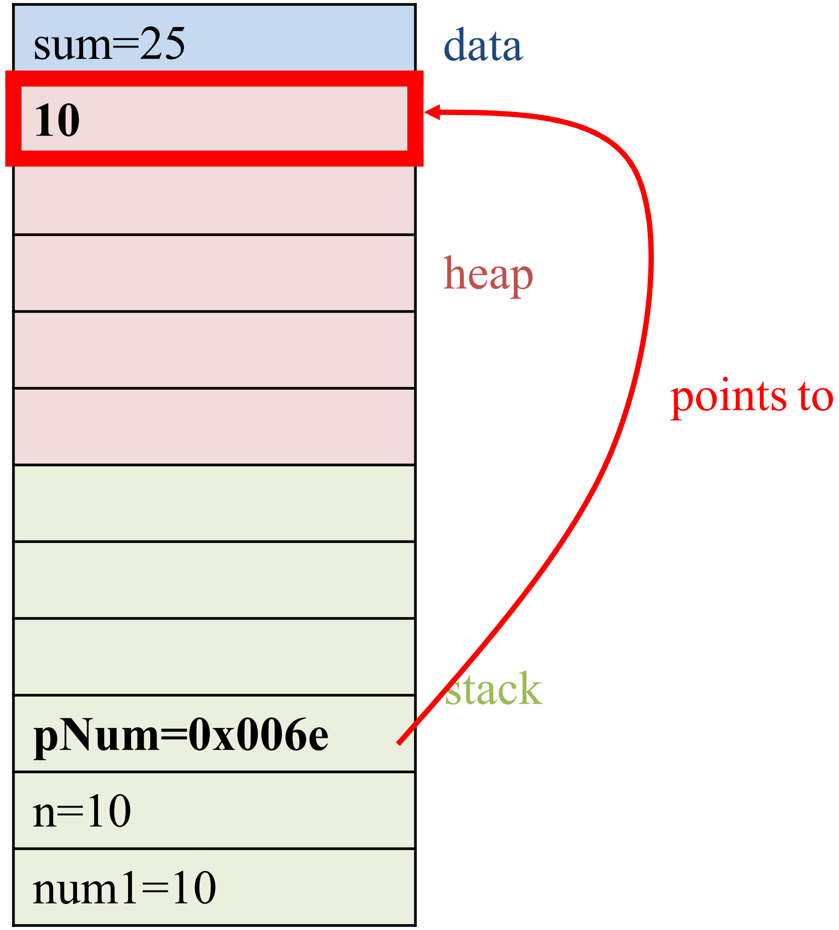
```
int sum=25;

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

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



Example - Memory Layout (Memory Leak) 2

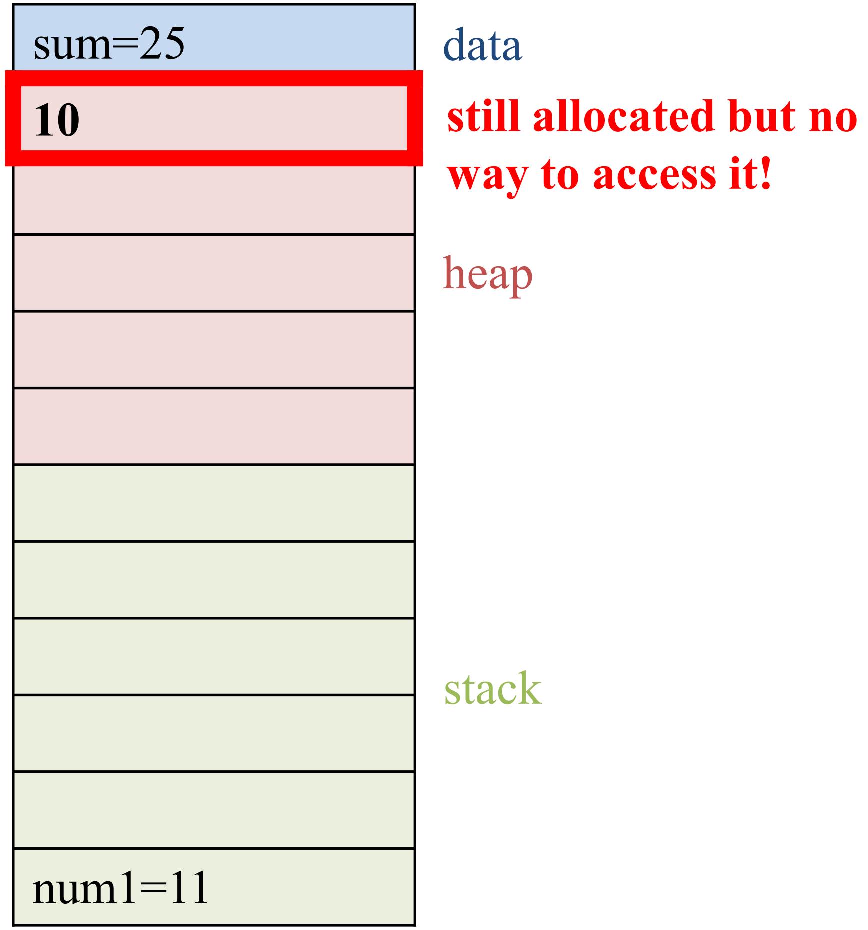
call

```
int sum=25;

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

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



Example - Memory Layout (Memory Leak) 3

```
int sum=25;

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

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



still allocated but no way to access it!

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 between Pointer & Reference

- A pointer is assigned by an address.

```
void func(int* pn) {...}

void main(){
    int a = 10;
    int* pa = &a;

    func(&a);
}
```

- A reference is initialized to an object (variable).

```
void func(int& rn) {...}

void main(){
    int b = 10;
    int& rb = b;

    func(b);
}
```

Differences between 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 between 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 refer to b, just copy value of b to a
cout << a << " " << b << " " << r << endl; // 2 2 2

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

Differences between 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++ uses "call-by-value" (or "pass-by-value")
    - Arguments are passed to functions by **copying values**
  - 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 a reference to a valid instance (whereas a pointer can be null).

```
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 of these 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;
}
```

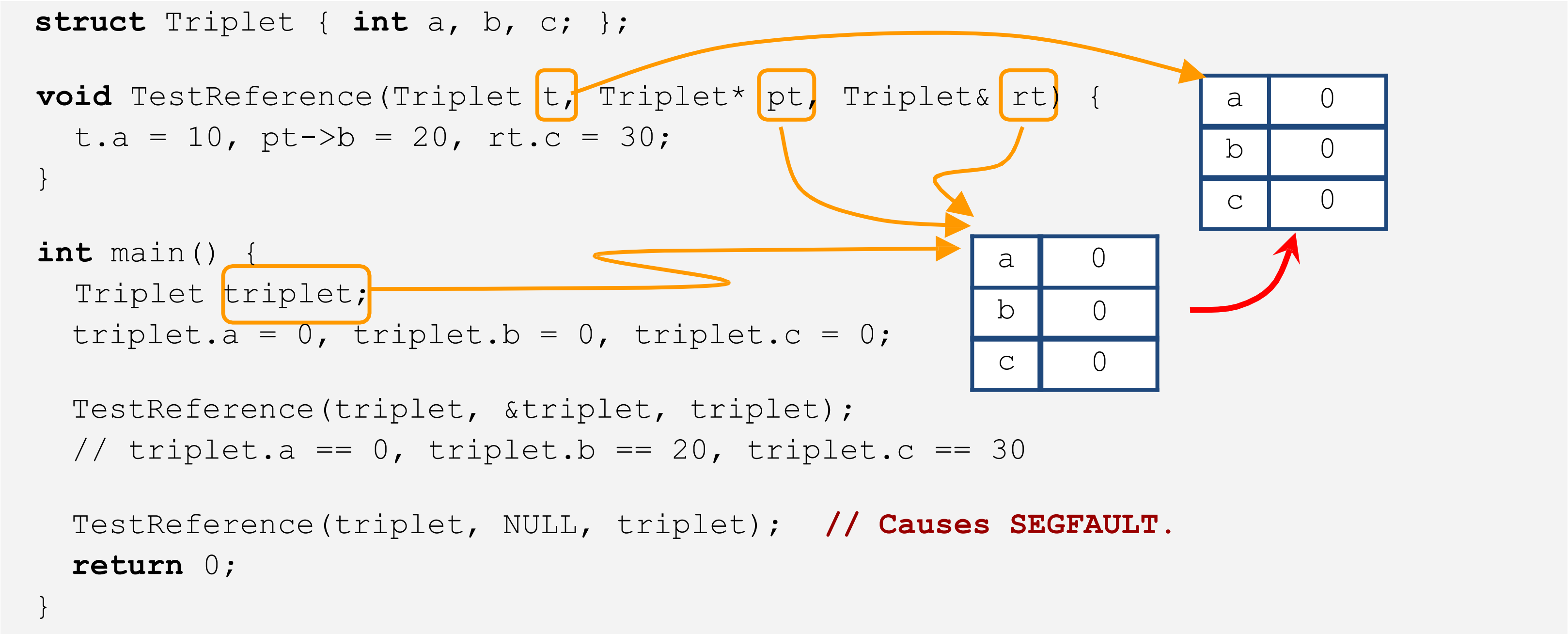
Recall: When to use Pointers in C?

- "Simulation" of call-by-reference in C
  - 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 a reference to a valid instance (whereas a pointer can be null).



```
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.

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

- Address-of operator

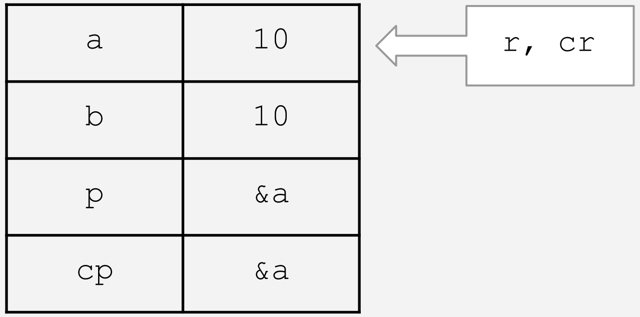
```
int a = 0;
int* pa = &a; // ‘&‘+[variable name]
```

- 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
```



a	10
b	10
p	&a
cp	&a

r, cr