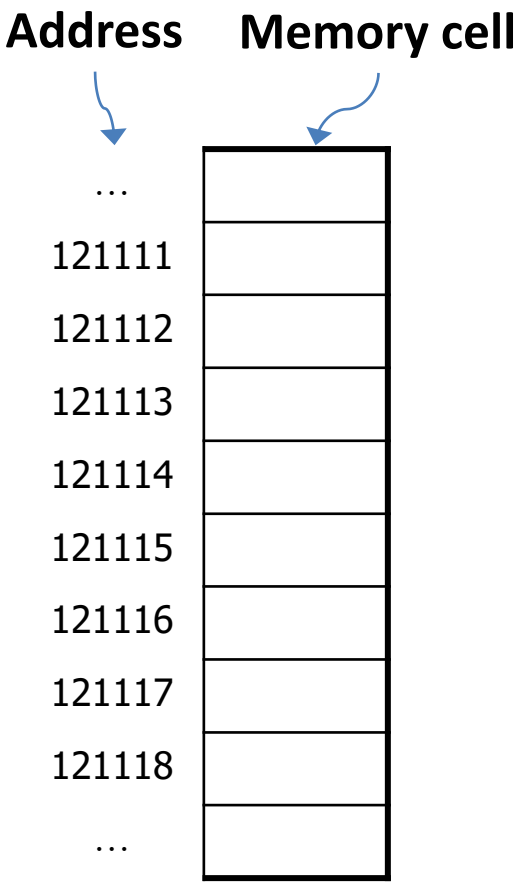
COMP2113 Programming Technologies / ENGG1340 Computer Programming II
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

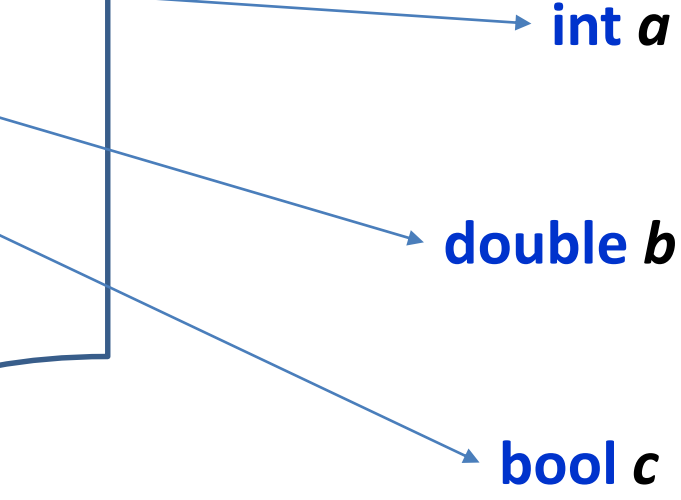
Memory address

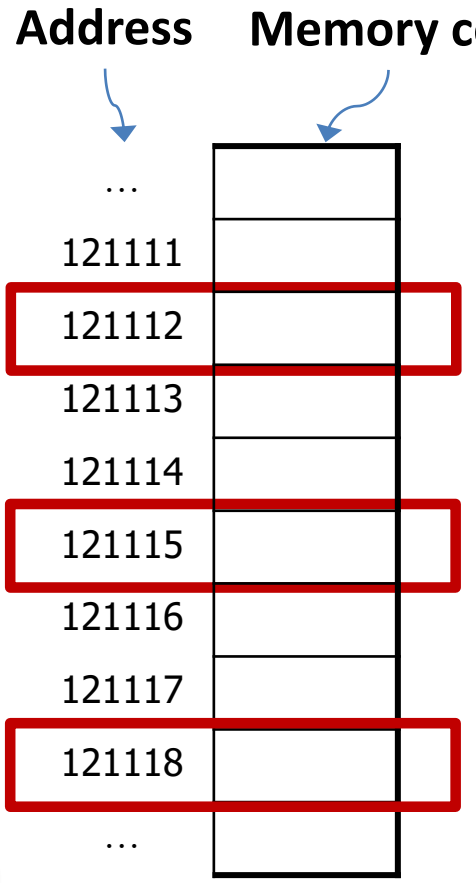


- The main memory is a collection of memory locations (or memory cells).
- Each memory location has a unique address.
- Thus, every variable you declared in your C++ program has:
  - The value in the memory cell.
  - The address of the variable.

Memory address

```
#include <iostream>
using namespace std;
int main(){
  int a;
  double b;
  bool c;

  return 0;
}
```





Address of variable:

The address of variable ***a*** is 121112.

The address of variable ***b*** is 121115.

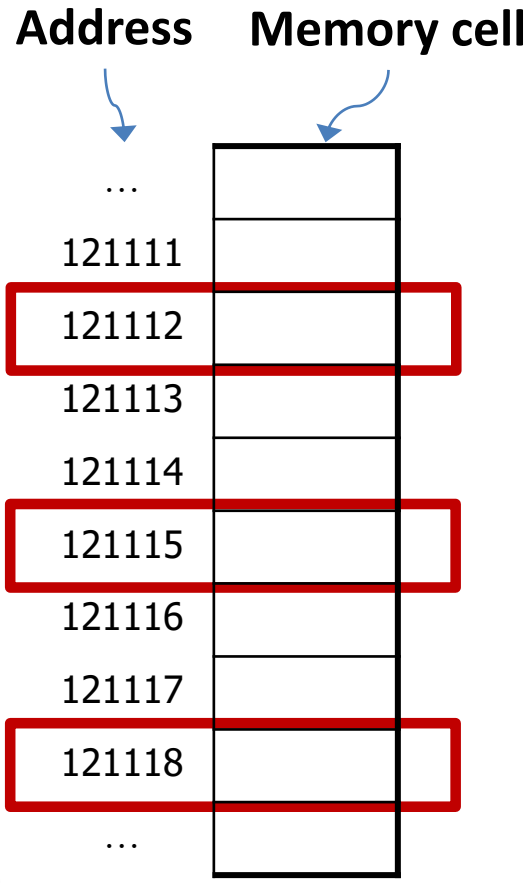
The address of variable ***c*** is 121118.

Memory address

```
#include <iostream>
using namespace std;
int main(){
  int a;
  double b;
  bool c;

  cout << &a << endl;
  cout << &b << endl;
  cout << &c << endl;
  return 0;
}
```

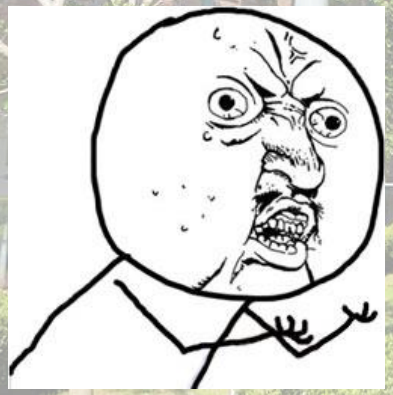
```
121112
121115
121118
```

Screen output
(The actual output may have different address values)



The address-of operator & returns the **address** of that variable.

Pointer Variable



Y U SO error-prone!?

1. Pointer variable

- A **pointer** is a variable that stores the **address value**.
- Declaration of a pointer variable:

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

struct student{
      string name;
      double assignment;
};

int main(){
   int *ip;
   double *dp;
   char *cp;
   student *sp;
  return 0;
}
```

type *variable-name;

Pointer that stores an **int** address.

Pointer that stores a **double** address.

Pointer that stores a **char** address.

Pointer that stores a **student** address.

1. Pointer variable

- **Type compatibility** – E.g., an **int** pointer variable can only store the address of an **int** variable.

```
#include <iostream>
using namespace std;
int main(){
  int x;
  int *adr_var;
  adr_var = & x;
  return 0;
}
```

adr_var is a pointer variable that stores **int** address, we need to assign the address of an **int** variable to it (i.e., **&*x***).

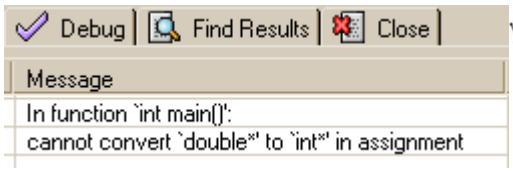
1. Pointer variable

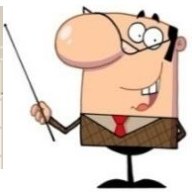
- **Type compatibility** – E.g., an **int** pointer variable can only store the address of an **int** variable.

```
#include <iostream>
using namespace std;
int main(){
   double x;
   int *adr_var;
   adr_var = & x;
   return 0;
}
```

This assignment statement is **wrong**!
&y is the address of the **double** value ***x***, so this value is a **double address.**
adr_var is an **int** pointer variable, which can only store an **int** address.

Compilation Error!!!





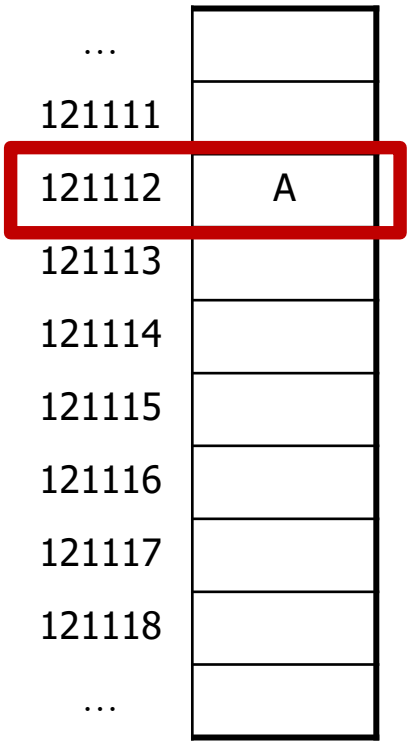
2. Dereference a pointer

- When there is a * before an address (Not in variable declaration), the expression refers to the **memory cell** by its address.

*** (121112)**

An address

The **memory cell** of the address (i.e., the cell storing 'A')

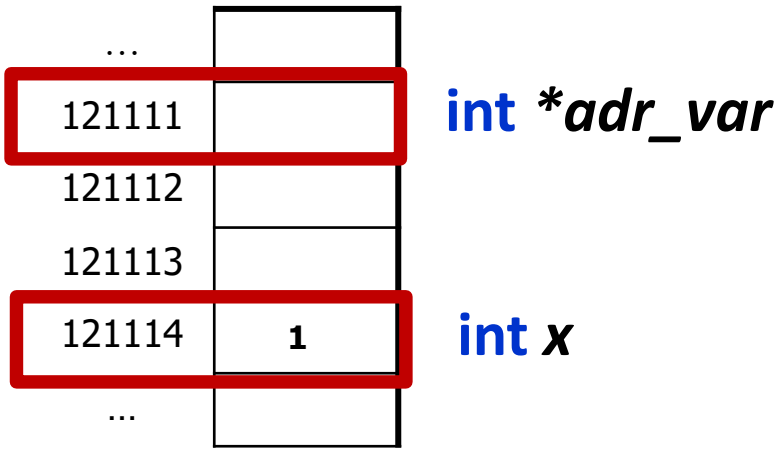


2. Dereference a pointer

```
...
int main(){
    int x = 1, * adr_var;
    adr_var = &x;
    cout << * adr_var  << endl;
    * adr_var = 44 * 2;
    cout << x << endl;
    return 0;
}
```

Declaration of variables

- Note that ***x*** is an **int** variable, initialized to 1.
- Since there is a **"*"** in the declaration of the variable ***adr_var***, this is a **pointer variable** that stores an **int** address.

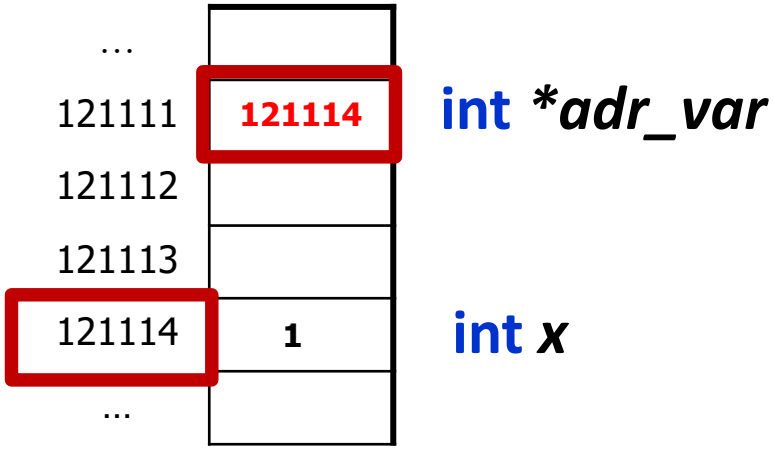


2. Dereference a pointer

```
...
int main(){
    int x = 1, * adr_var;
    adr_var = &x;
    cout << * adr_var  << endl;
    * adr_var = 44 * 2;
    cout << x << endl;
    return 0;
}
```

Assign address to a pointer

- "***&x***" refers to the **address** of the variable ***x***. In this example, it is 121114.

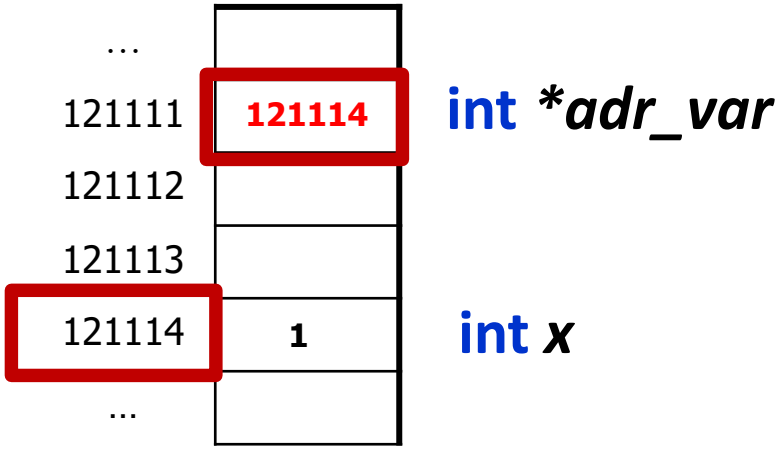


2. Dereference a pointer

```
...
int main(){
    int x = 1, * adr_var;
    adr_var = &x;
    cout << * adr_var  << endl;
    * adr_var = 44 * 2;
    cout << x << endl;
    return 0;
}
```

Dereference a pointer

- ***adr_var*** stores the address value 121114.
- ****adr_var*** means the memory cell with address 121114.



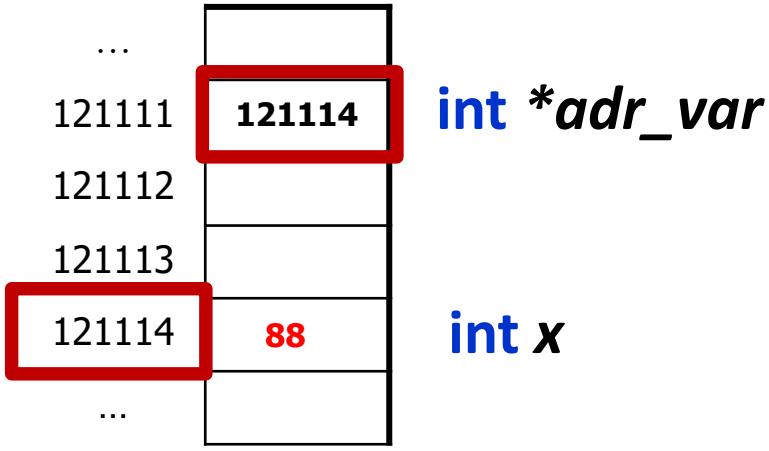
Screen output

2. Dereference a pointer

```
...
int main(){
    int x = 1, * adr_var;
    adr_var = &x;
    cout << * adr_var  << endl;
    * adr_var = 44 * 2;
    cout << x << endl;
    return 0;
}
```

An expression

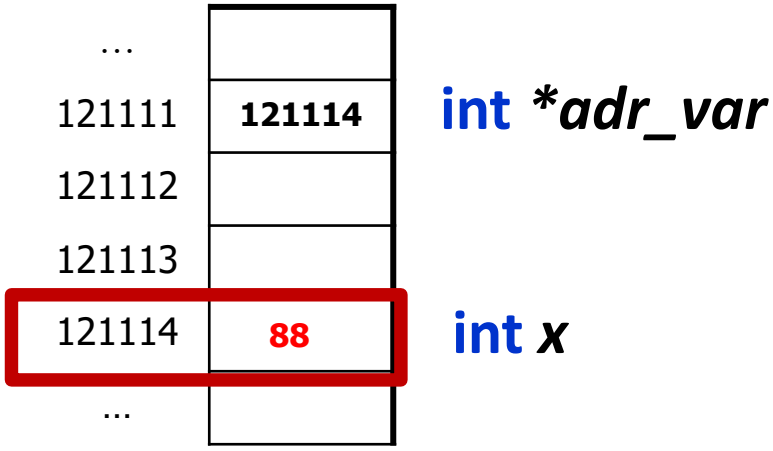
- The left part of the expression is ***adr_var**, which means *(121114). The memory cell with address 121114.



Screen output

2. Dereference a pointer

```
...
int main(){
    int x = 1, * adr_var;
    adr_var = &x;
    cout << * adr_var  << endl;
    * adr_var = 44 * 2;
    cout << x << endl;
    return 0;
}
```




```
1
88
```

Screen output

2. Dereference a pointer

```
...
int main(){
    int x = 1, * adr_var;
    adr_var = &x;
    cout << * adr_var << endl;
    * adr_var = 44 * 2;
    cout << x << endl;
    return 0;
}
```

Pointer declaration

- **When * is in declaration**, it specifies pointer variable, it can store an **int** address.

Dereference pointer

- **When * is before an address**, it is referring to the memory cell with that address.

Arithmetic operator

Example 1

```
...
struct Student{
        int UID;
        double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

Structure Student

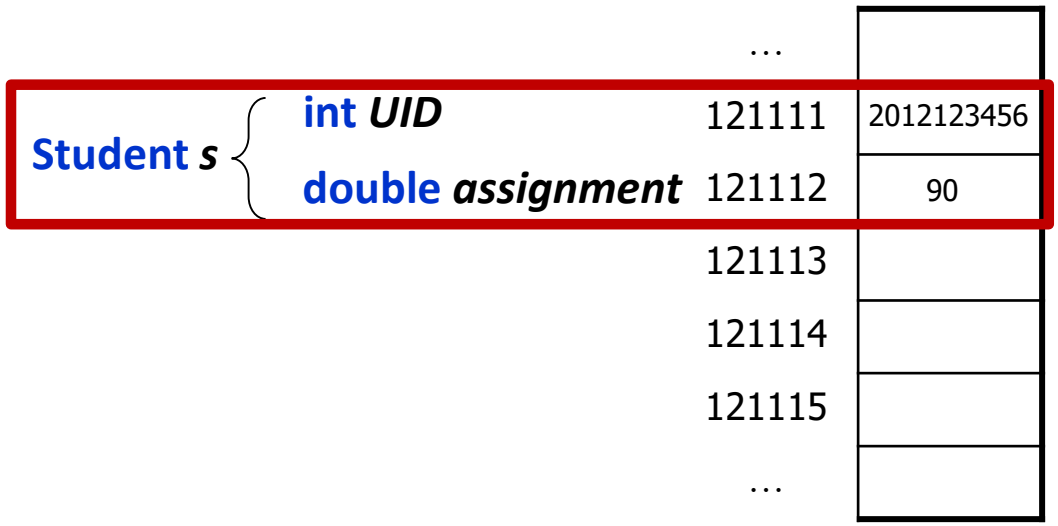
- Declare a **struct Student**.
  A student has two **member variables**:
  1. **int** UID
  2. **double** assignment

Example 1

```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

Declare a student

- Declare a **Student** struct variable ***s***. The computer allocates memory slots for storing the ***UID*** and ***assignment*** values of the **Student** struct variable.

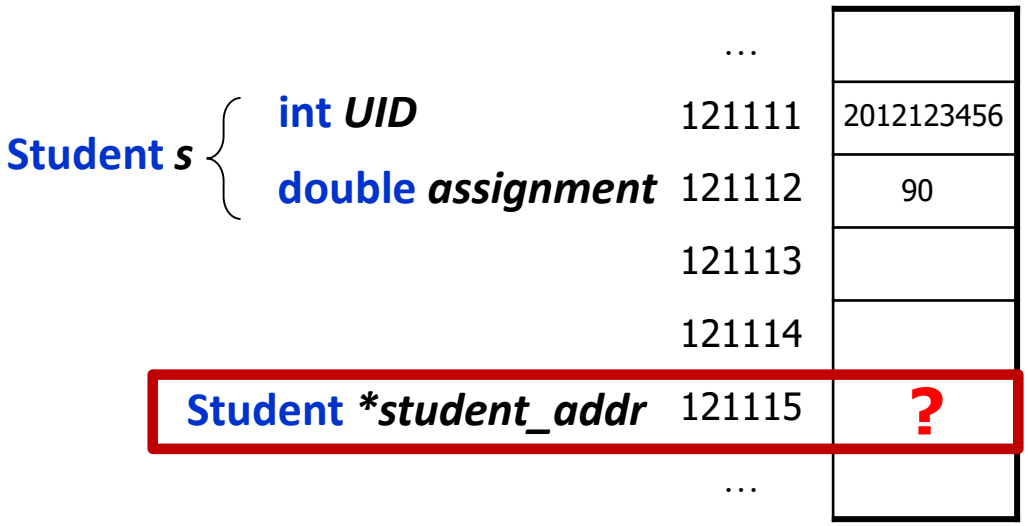


Example 1

```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

Pointer to student

- Declare a pointer variable ***student_addr.***
  student_addr can stores only the address of a **Student** variable.

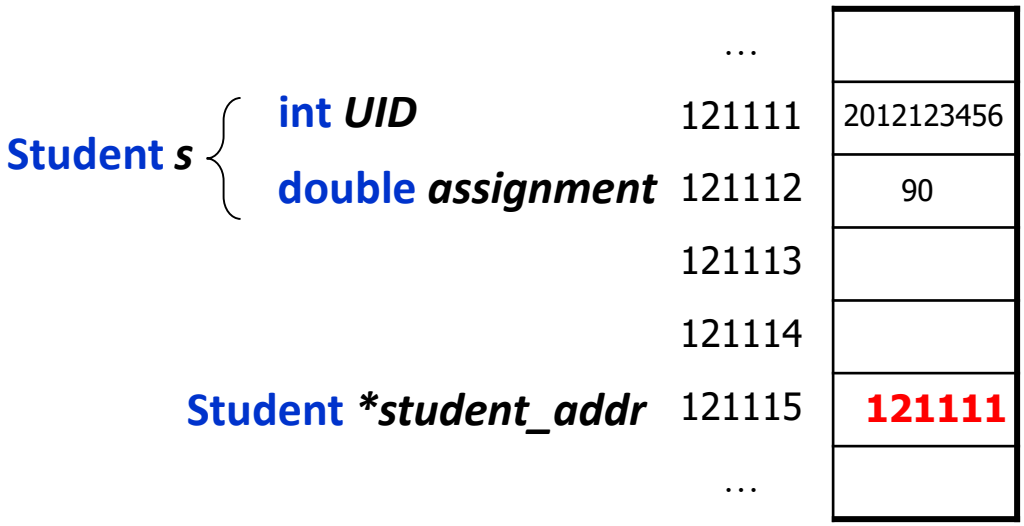


Example 1

```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

Type compatible

- ***s*** is a variable for storing Student value. Thus, its address can be assigned to ***student_addr***.

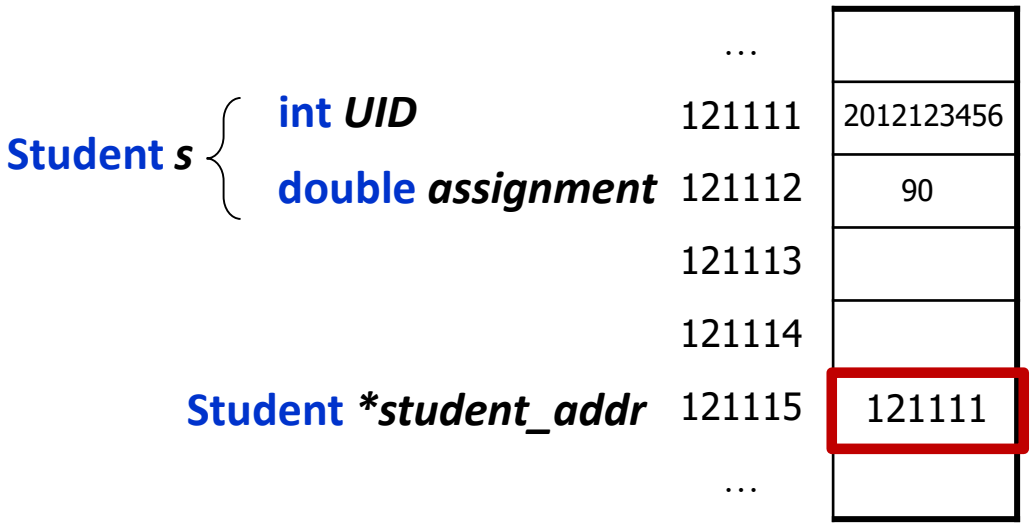


Example 1

```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

(* **student_addr**) .*UID*

121111



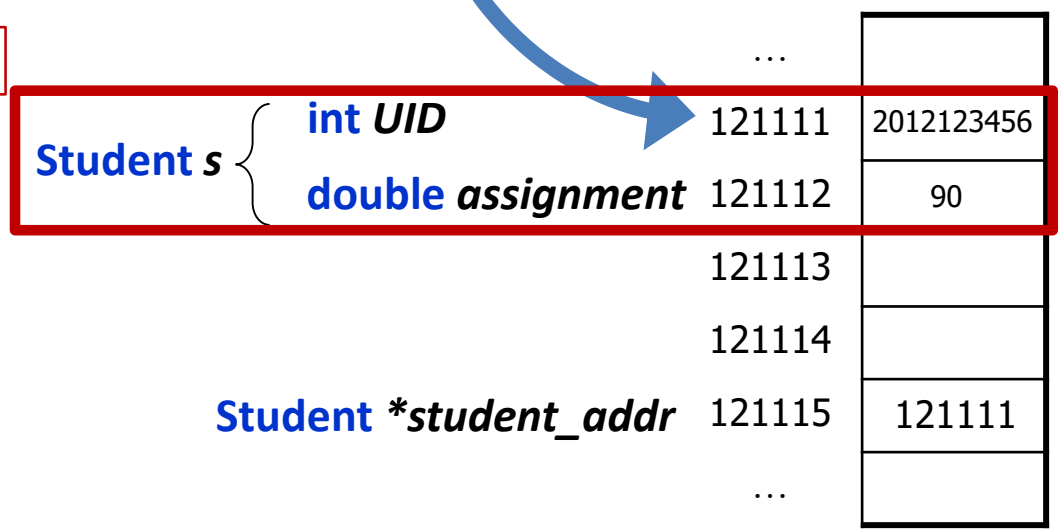
Example 1

```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

(* **student_addr**) .UID

121111

The student **struct**



Example 1

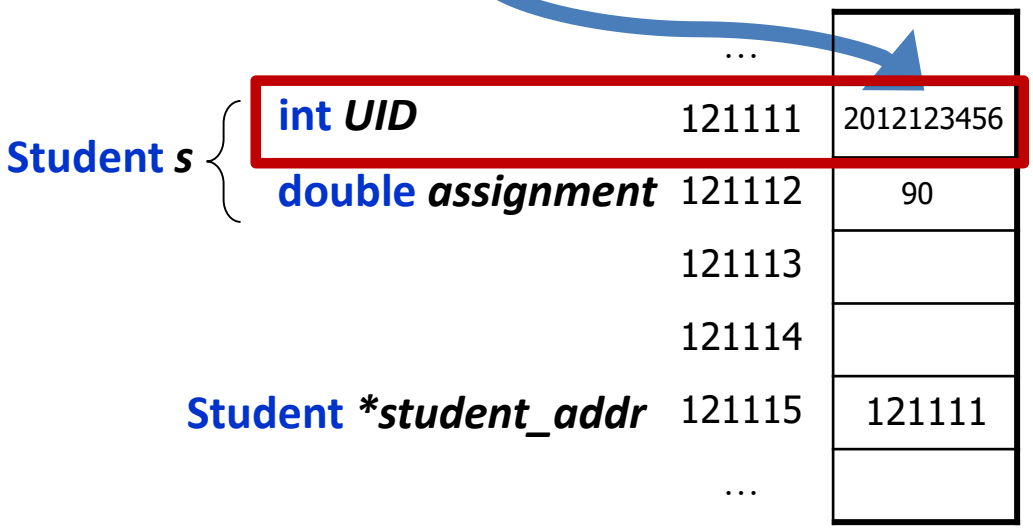
```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

(* **student_addr**) .***UID***

121111

The student **struct**

The UID value of the student

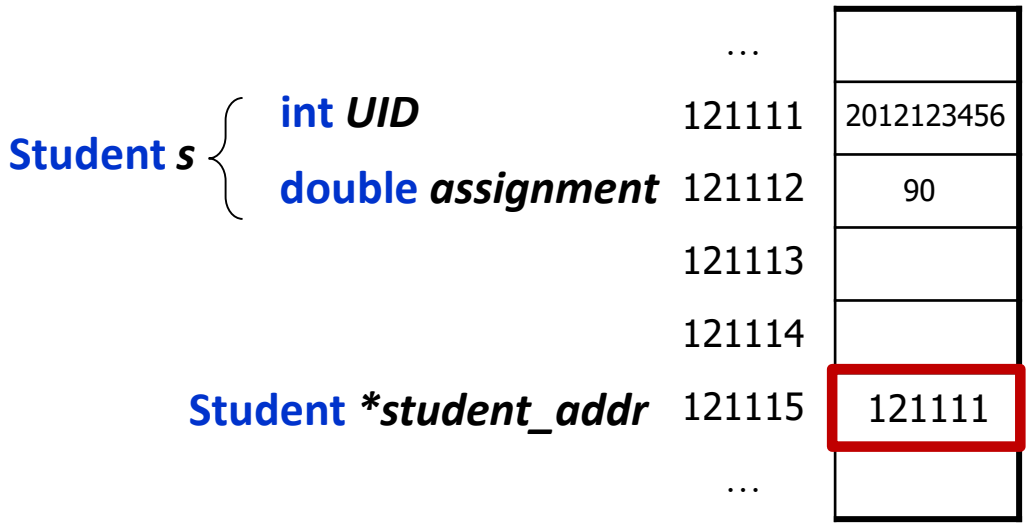


Example 1

```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

(* **student_addr**) .*assignment*

121111



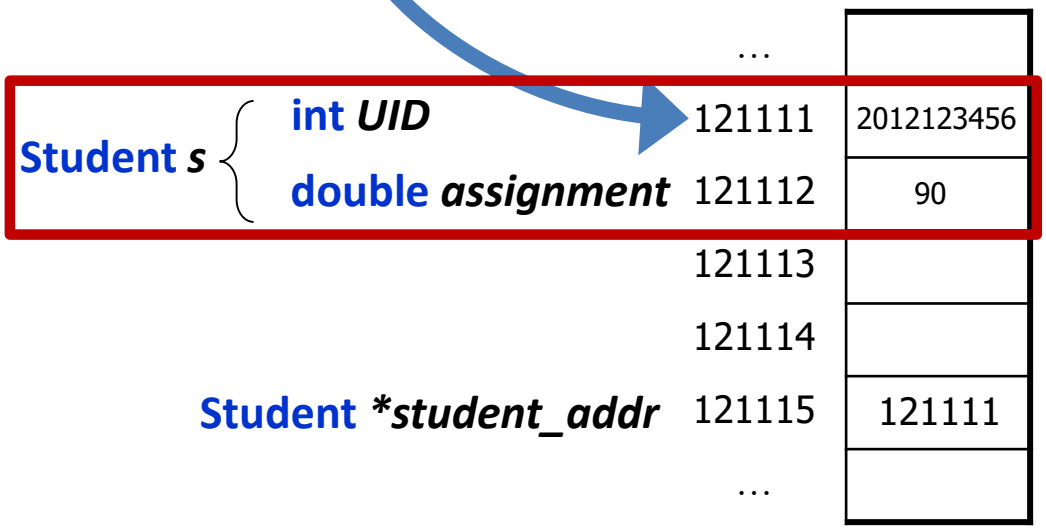
Example 1

```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

(* student_addr) .assignment

121111

The student struct



Example 1

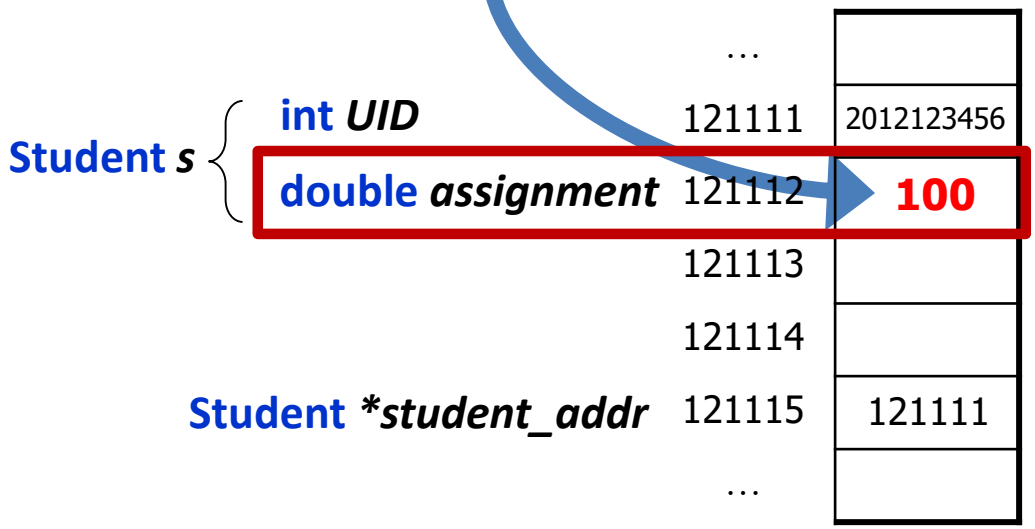
```
...
struct Student{
      int UID;
      double assignment;
};

int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;

  cout << (*student_addr).UID << endl;

  (* student_addr ).assignment = 100;

  return 0;
}
```

(* **student_addr**) .***assignment***

121111

The student **struct**

The assignment value of the student



3. Pointer and struct / class

- **-> is the member access operator.**
- Roughly speaking, if ***a*** is a pointer to a structure, you can use ***a*->*b*** to access the member variable ***b***.

```
...
struct Student{
    int UID;
    double assignment;
};
int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;
   cout << (*student_addr).UID << endl;
   (* student_addr ).assignment = 100;
  return 0;
}
```

a->b ≡ (*a).b

Pointer to a struct/ object

Member variable

(***student_addr***).***UID***

is the same as:

student_addr->UID

3. Pointer and struct / class

- **-> is the member access operator.**
- Roughly speaking, if ***a*** is a pointer to a structure, you can use ***a*->*b*** to access the member variable ***b***.

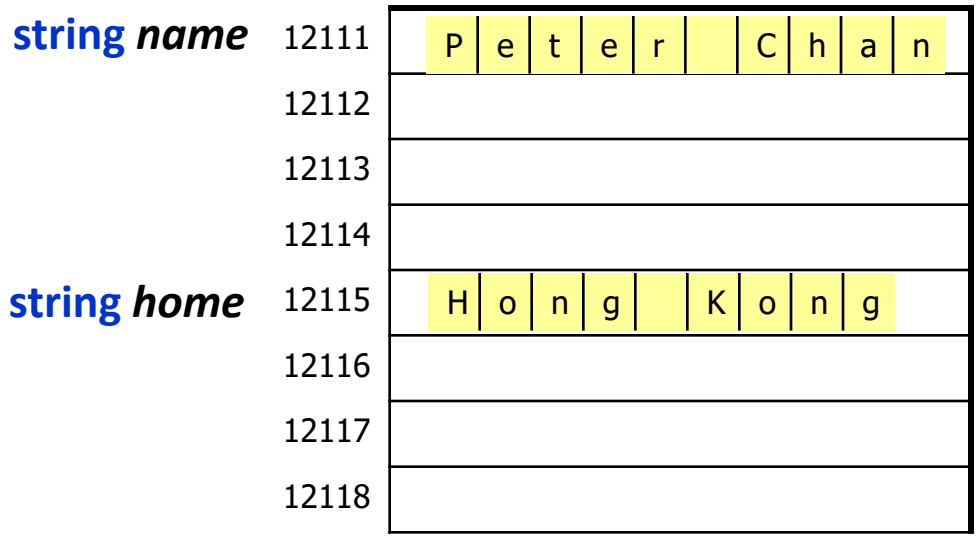
```
...
struct Student{
        int UID;
        double assignment;
};
int main(){
  Student s = {2012123456, 90};
  Student *student_addr = &s;
   cout << (*student_addr).UID << endl;
  (* student_addr ).assignment = 100;
  return 0;
}
```

≡

```
...
struct Student{
        int UID;
        double assignment;
};
int main(){
    Student s = {2012123456, 90};
    Student *student_addr = &s;
    cout << student_addr->UID << endl;
    student_addr ->assignment = 100;
    return 0;
}
```

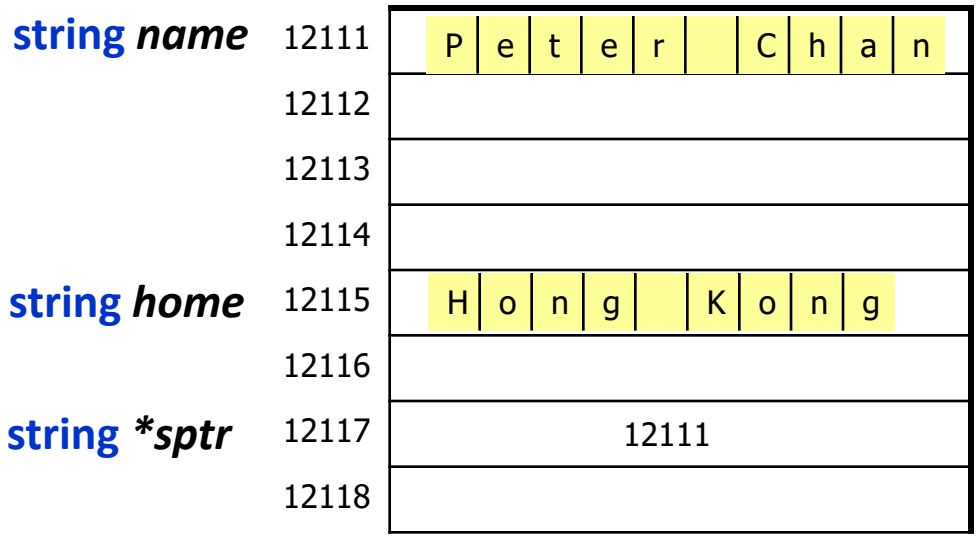
Example 2

```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```



Example 2

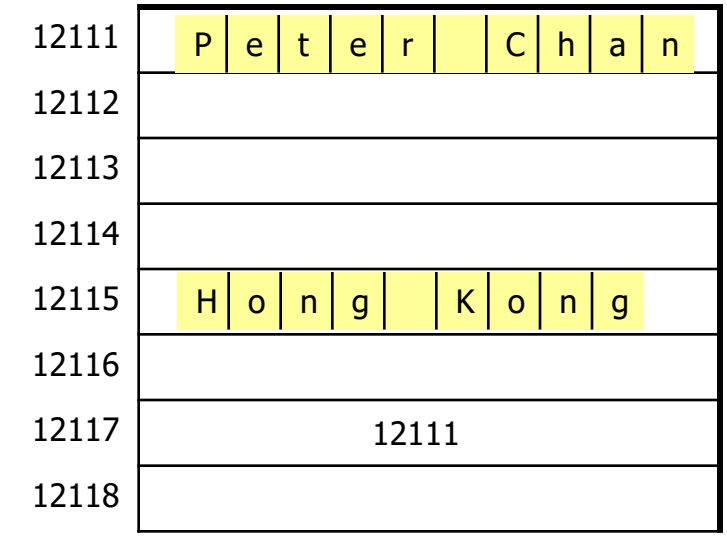
```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```

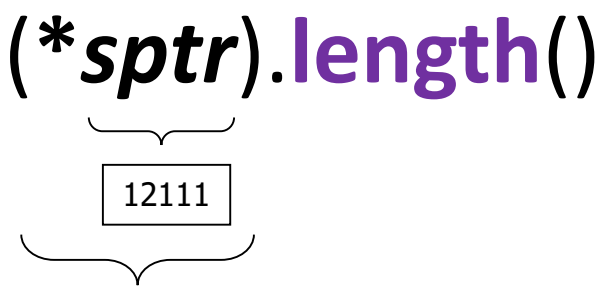


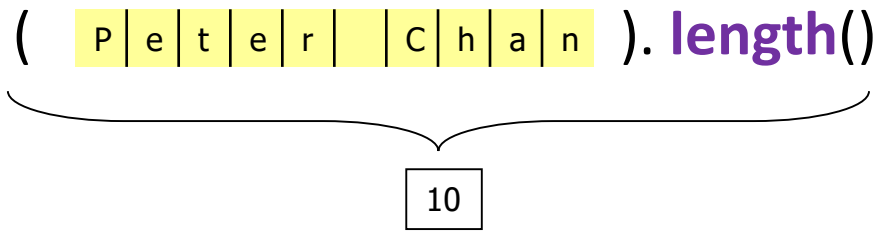
Example 2

```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```

Screen output



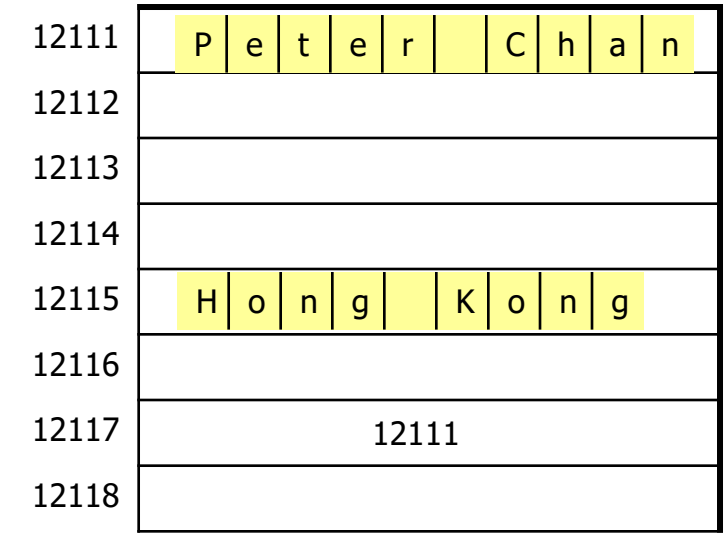


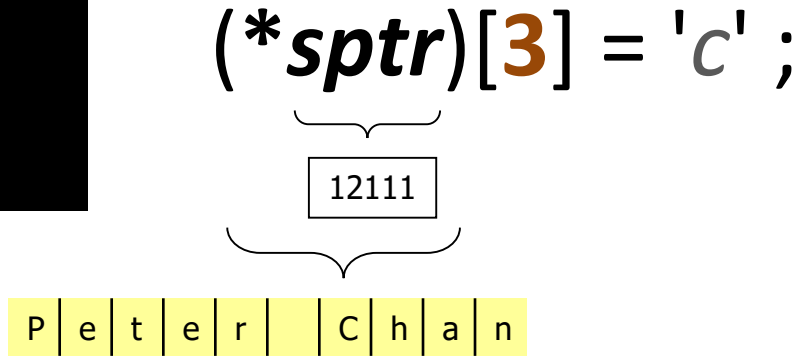


Example 2

```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```

Screen output

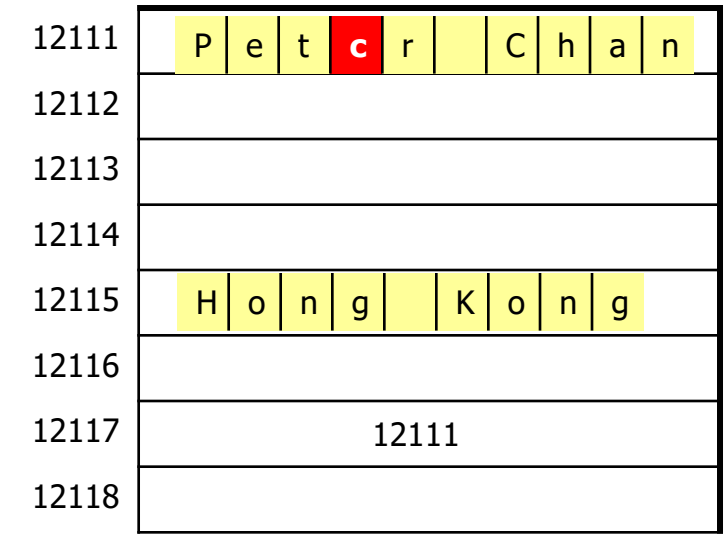


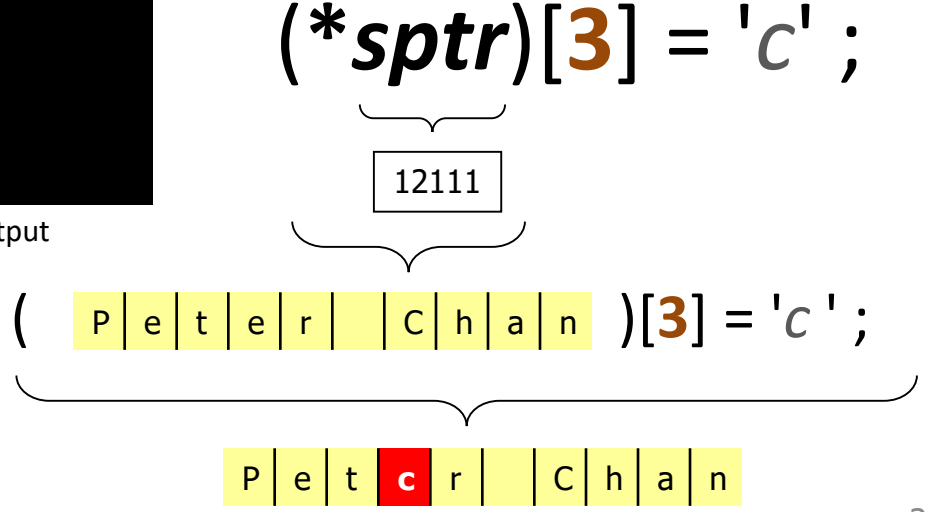


Example 2

```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```

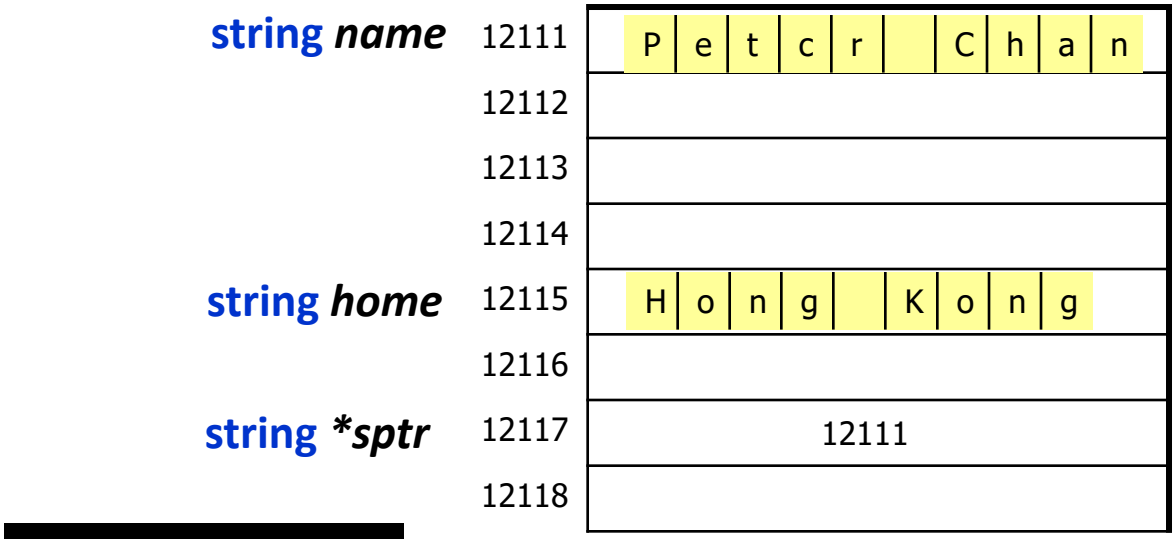
Screen output





Example 2

```
#include <iostream>
using namespace std;
int main(){
   string name ("Peter Chan");
   string home ("Hong Kong");
   string * sptr = &name;

   cout << (*sptr).length() << endl;
   (*sptr)[3] = 'c' ;
   cout << (*sptr).substr(0, 5) << endl;

   sptr = &home;
   cout << (*sptr).length() << endl;
   (*sptr)[3] = 'c';
   cout << (*sptr).substr(0, 4) << endl;

   return 0;
}
```




```
10
Petcr
```

Screen output

(*sptr).substr(0, 5);

12111

(P e t c r C h a n).substr(0, 5);

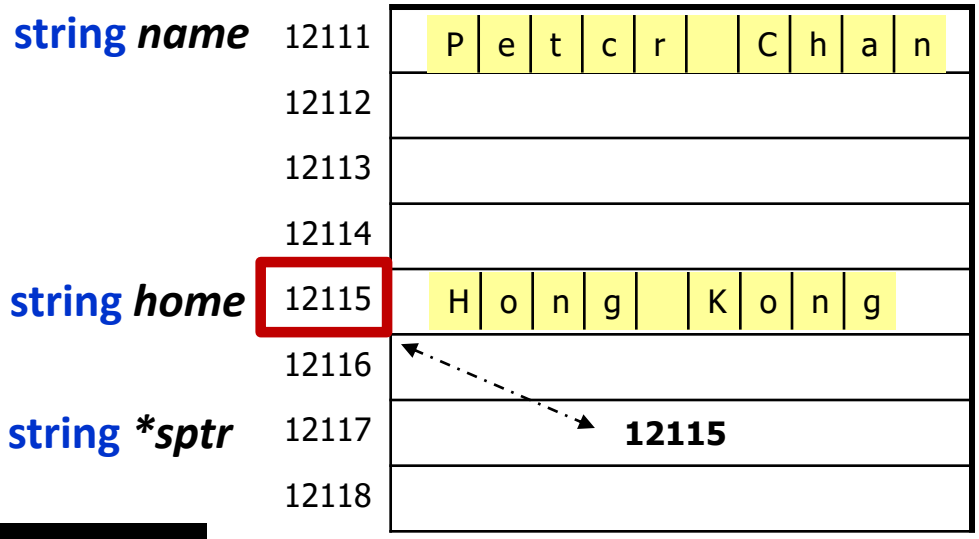
Petcr

Example 2

```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```

```
10
Petcr
```

Screen output

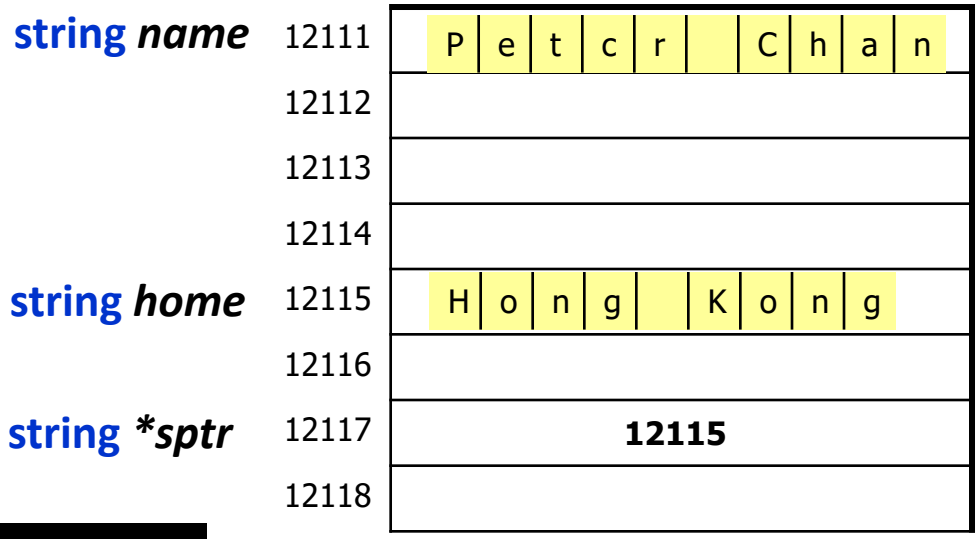


Example 2

```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```

```
10
Petcr
9
```

Screen output

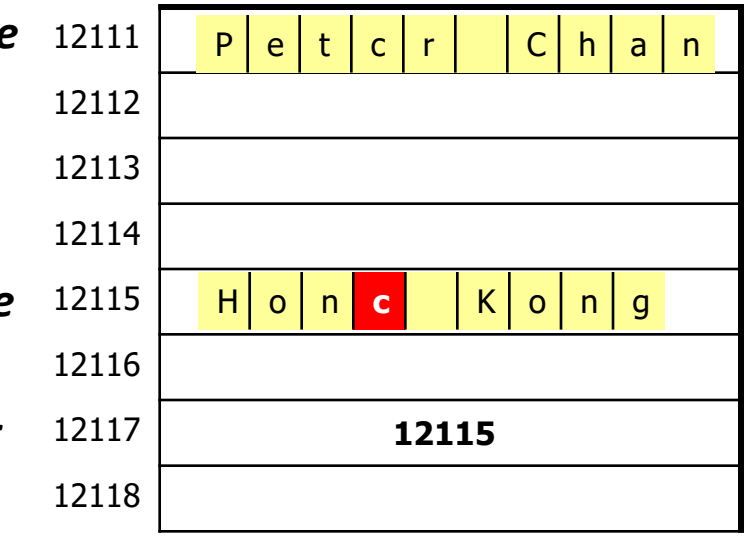


Example 2

```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```

```
10
Petcr
9
```

Screen output

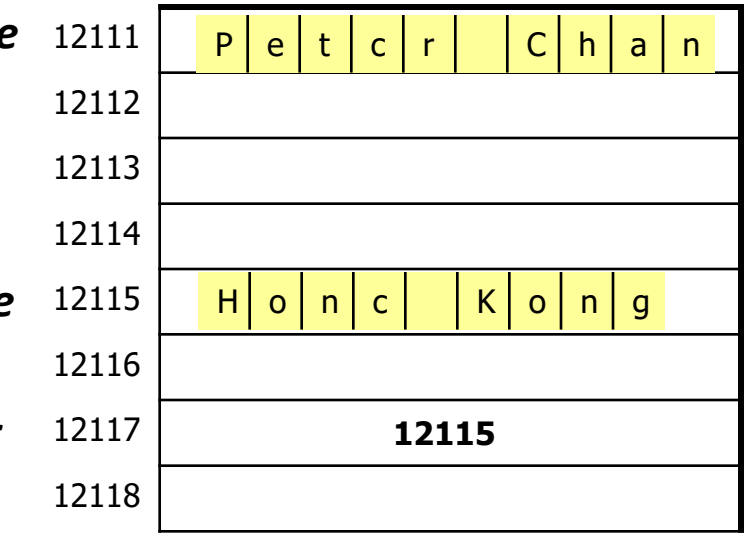


Example 2

```
#include <iostream>
using namespace std;
int main(){
    string name ("Peter Chan");
    string home ("Hong Kong");
    string * sptr = &name;

    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c' ;
    cout << (*sptr).substr(0, 5) << endl;

    sptr = &home;
    cout << (*sptr).length() << endl;
    (*sptr)[3] = 'c';
    cout << (*sptr).substr(0, 4) << endl;

    return 0;
}
```

```
10
Petcr
9
Honc
```

Screen output



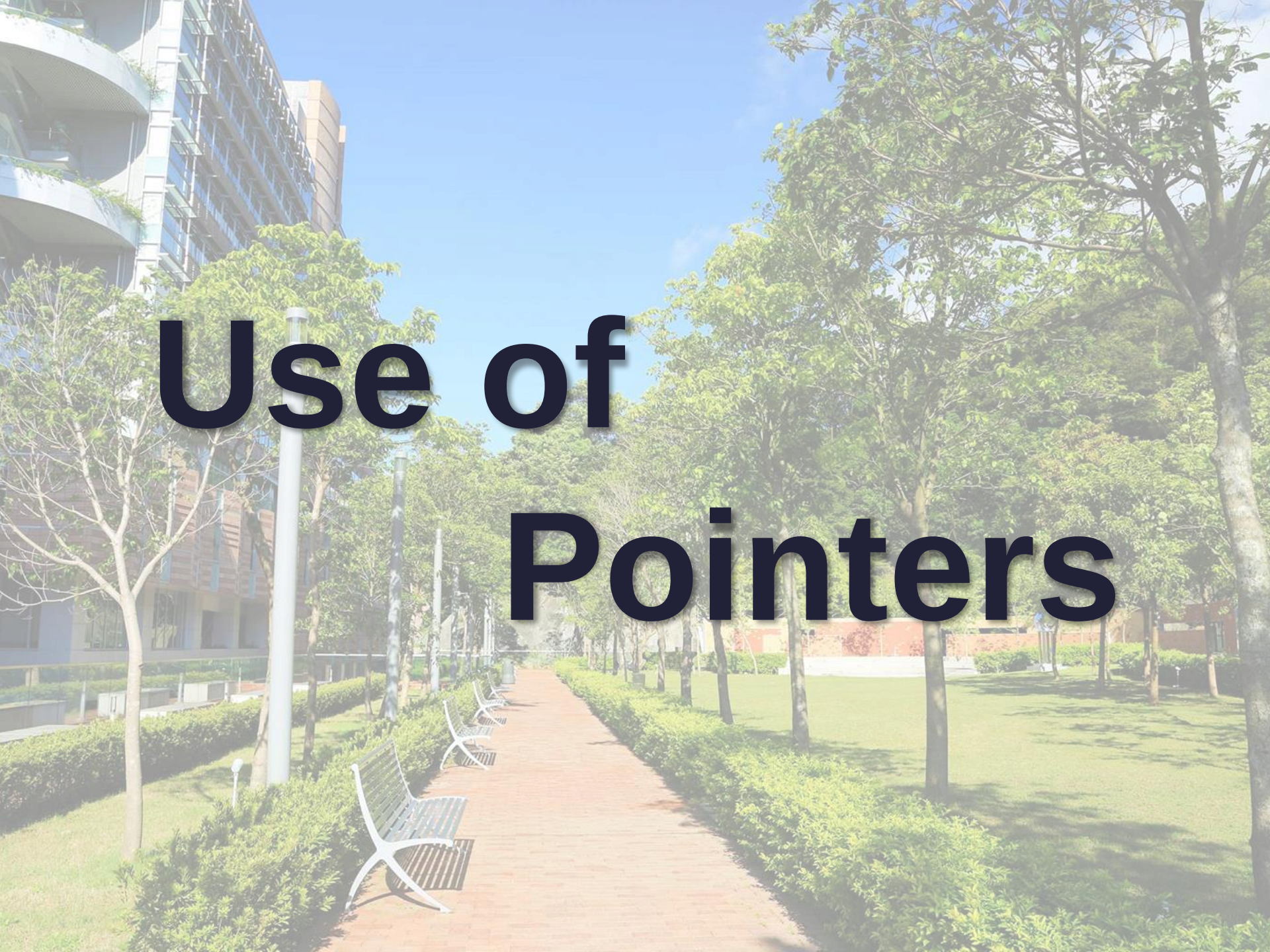
NULL

- You can initialize pointer variables as **NULL** (stores no address value).

```
...
struct Student{
      int UID;
      double assignment;
};
int main(){
    Student *s = NULL;
    if (s == NULL) {
          cout << "This is a NULL pointer" ;
    }
    return 0;
}
```

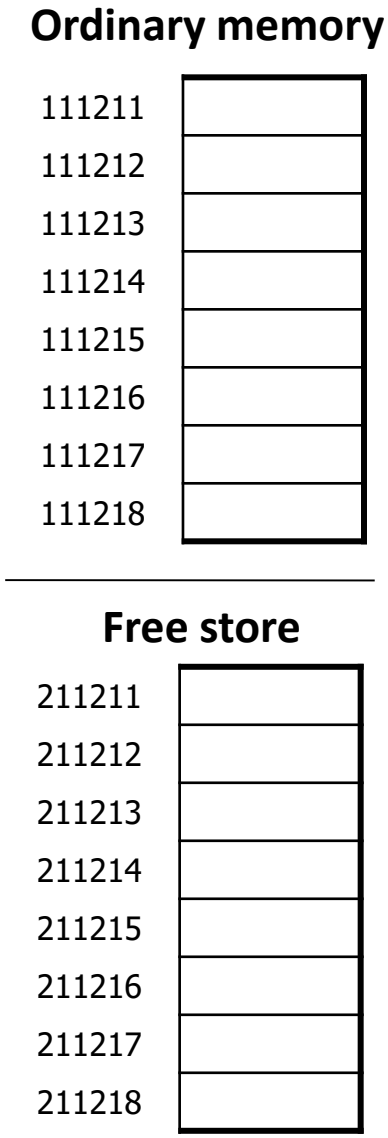
NULL Pointer initialization

- **NULL** is a special value that mean nothing is stored in the pointer (no address).


Use of
Pointers

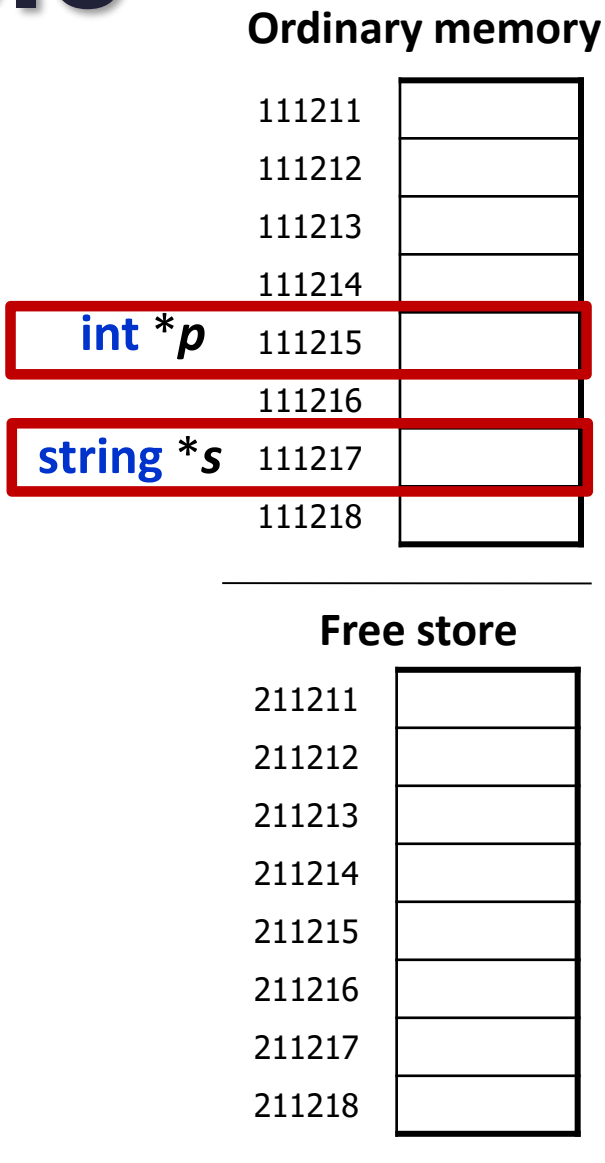
1. Dynamic variable

- When the program is put into execution, variables declared in the program such as **int**, **double**, **char**, **string**, **array**, **struct**, **pointer** …etc are stored in the **ordinary memory**.
- There are also **free store memory** that we can use for **dynamic memory allocation**.



1. Dynamic variable

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

int main(){
  int *p;
  string *s;

  p = new int;
  s = new string;

  *p = 8;
  *s = "dragon";

  return 0;
}
```



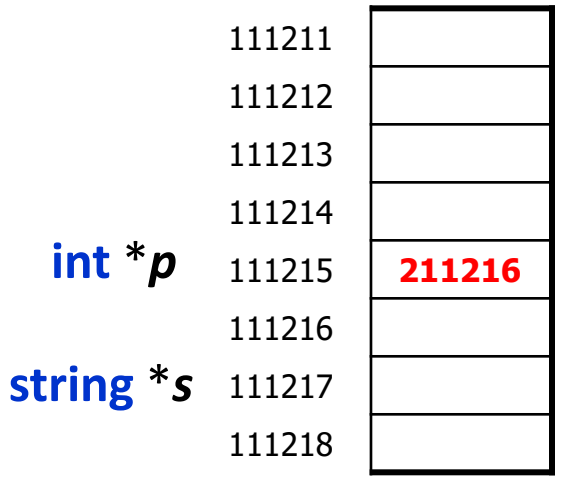
1. Dynamic variable

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

int main(){
  int *p;
  string *s;

  p = new int;
  s = new string;

  *p = 8;
  *s = "dragon";

  return 0;
}
```

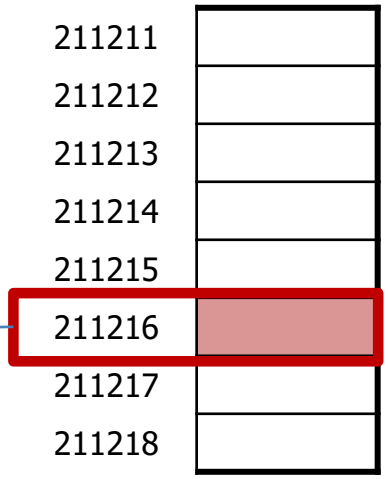
The keyword "new"

- Reserves cell from the free store for storing **int** value.
- After assigning the memory cell, the **new int** **returns the address of this newly reserved cell.**

Ordinary memory



Free store



***p* = new int;**

211216

1. Dynamic variable

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

int main(){
  int *p;
  string *s;

  p = new int;
  s = new string;

  *p = 8;
  *s = "dragon";

  return 0;
}
```

The keyword "new"

- Reserves cell from the free store for storing **string** value.
- After assigning the memory cell, the **new string returns the address of this newly reserved cell.**

s = new string;

211213

Ordinary memory

111211	
111212	
111213	
111214	
111215	211216
111216	
111217	211213
111218	

int *p

string *s

Free store

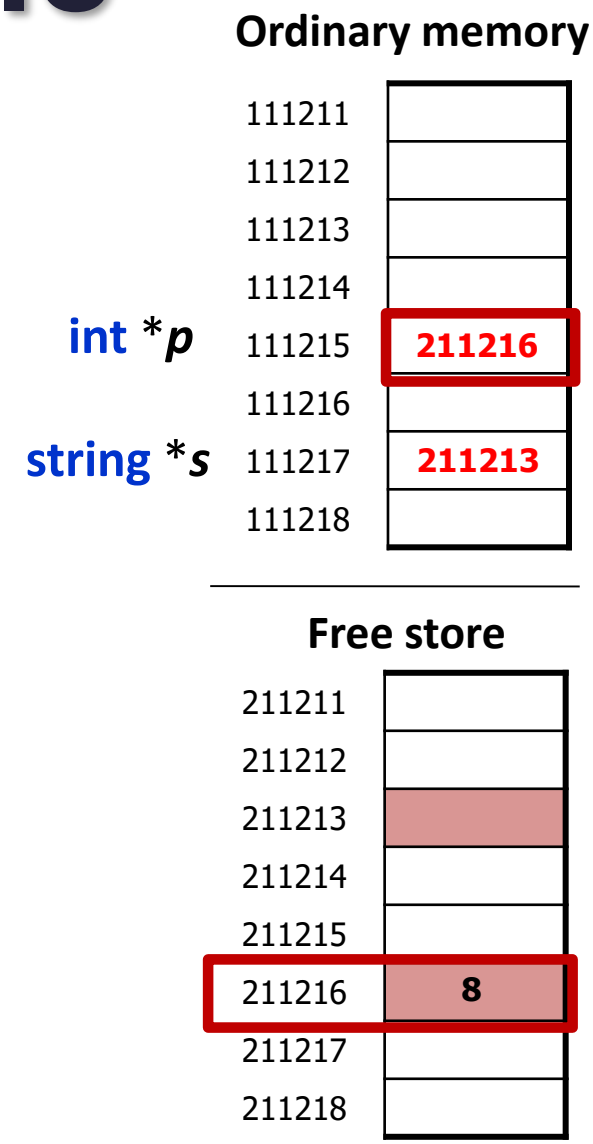
211211	
211212	
211213	
211214	
211215	
211216	
211217	
211218	

1. Dynamic variable

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

int main(){
  int *p;
  string *s;

  p = new int;
  s = new string;

  *p = 8;
  *s = "dragon";

  return 0;
}
```

Dereference pointer *p*

- ***p** means the memory cell with address 211216.
- This expression stores the value 8 to the memory cell with address 211216.

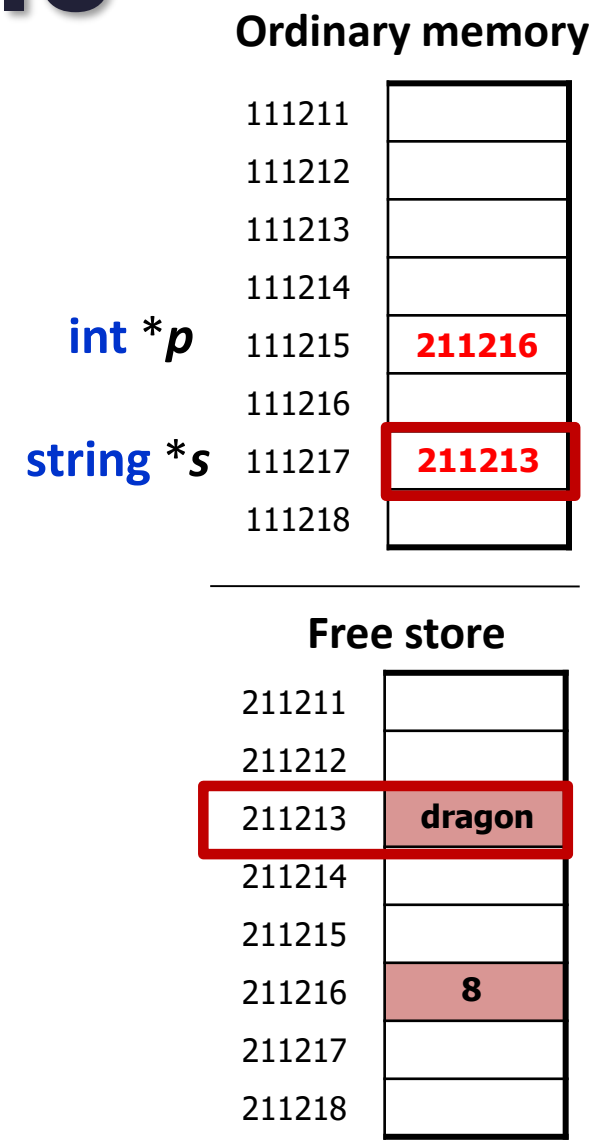


1. Dynamic variable

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

int main(){
  int *p;
  string *s;

  p = new int;
  s = new string;

  *p = 8;
  *s = "dragon";

  return 0;
}
```

Dereference pointer *s*

- ***s** means the memory cell with address 211213.
- This expression stores the value “dragon” to the memory cell with address 211213.



2. Pointers and functions

- **You can use pointers as function parameters.**

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

void swap (int * x, int  * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (&a, &b);
    cout << a << " " << b;
    return 0;
}
```

Use pointers as function parameters

- We can use pointers as function parameters, the effect is like pass by reference. (i.e., the swap works!)

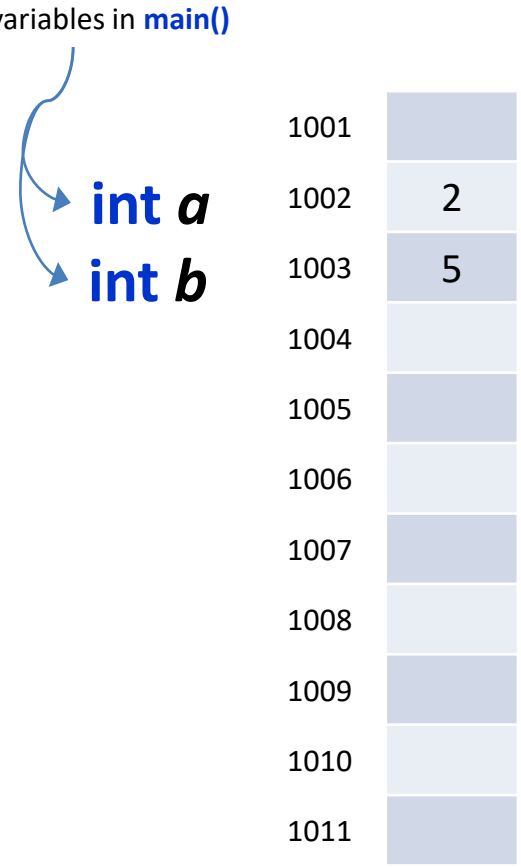
Calling the function

- To call the function, you need to pass the **addresses** to the pointer parameters (with correct address type).

2. Pointers and functions

- **Passing address to the function will have the same effect as pass by reference.**

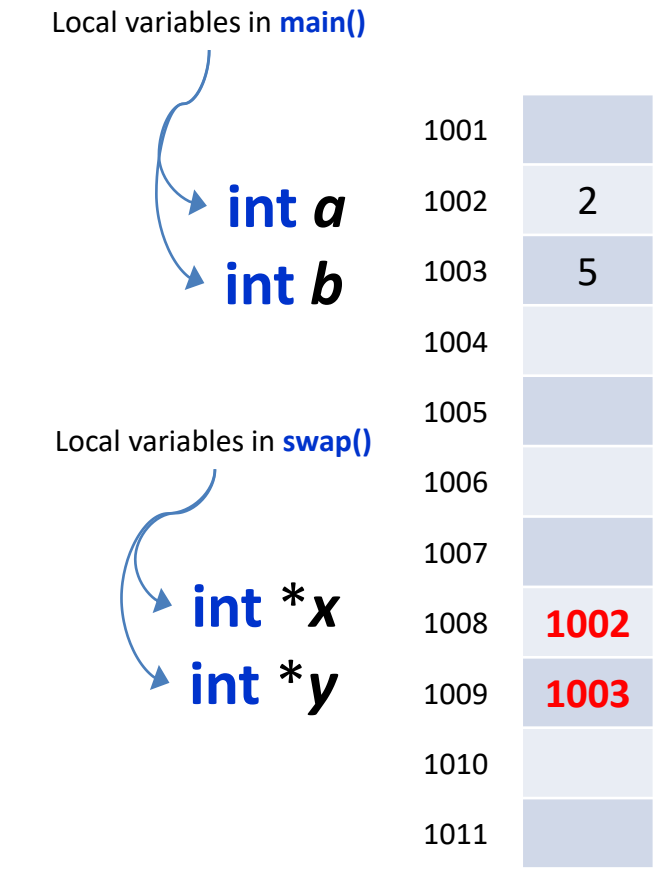
```
#include <iostream>
using namespace std;

void swap (int * x, int  * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (&a, &b);
    cout << a << " " << b;
    return 0;
}
```



2. Pointers and functions

- **Passing address to the function will have the same effect as pass by reference.**

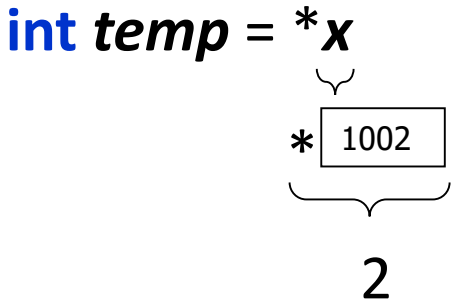
```
#include <iostream>
using namespace std;

void swap (int * x, int  * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (&a, &b);
    cout << a << " " << b;
    return 0;
}
```

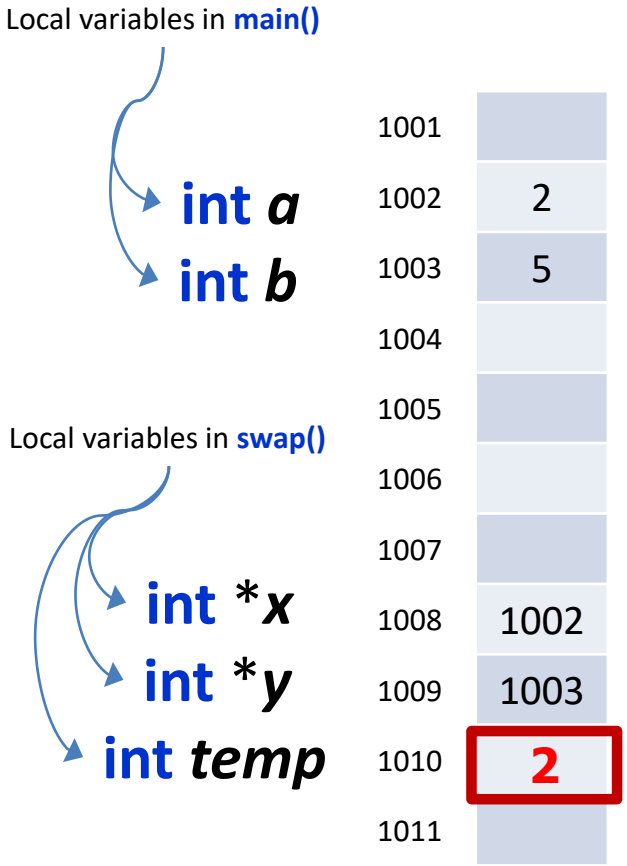


2. Pointers and functions

- **Passing address to the function will have the same effect as pass by reference.**

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

void swap (int * x, int  * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (&a, &b);
    cout << a << " " << b;
    return 0;
}
```

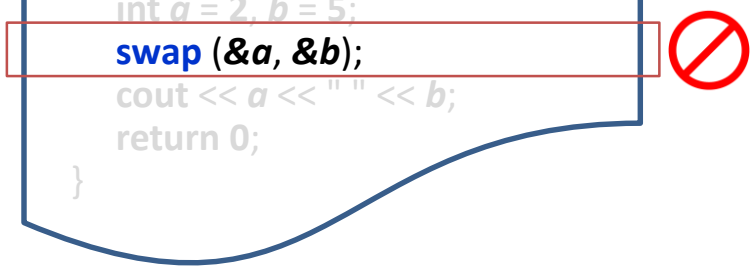


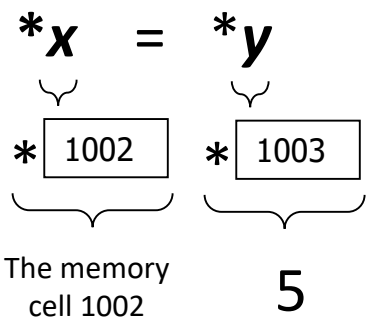


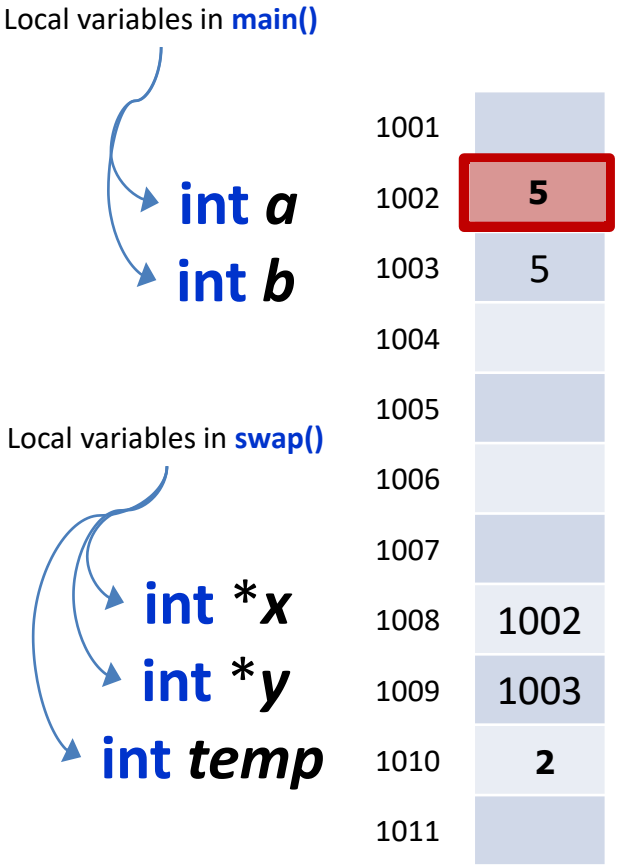
2. Pointers and functions

- **Passing address to the function will have the same effect as pass by reference.**

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

void swap (int * x, int  * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (&a, &b);
    cout << a << " " << b;
    return 0;
}
```



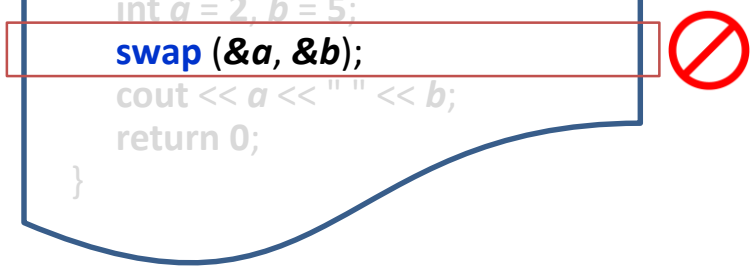


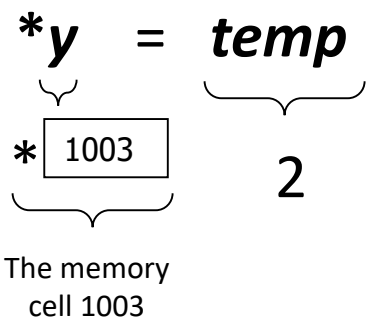


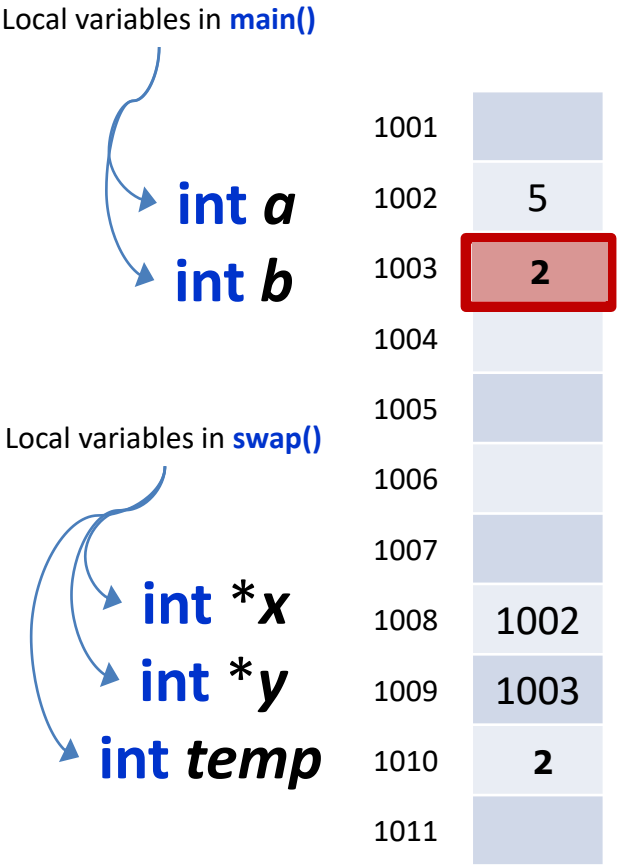
2. Pointers and functions

- **Passing address to the function will have the same effect as pass by reference.**

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

void swap (int * x, int  * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (&a, &b);
    cout << a << " " << b;
    return 0;
}
```







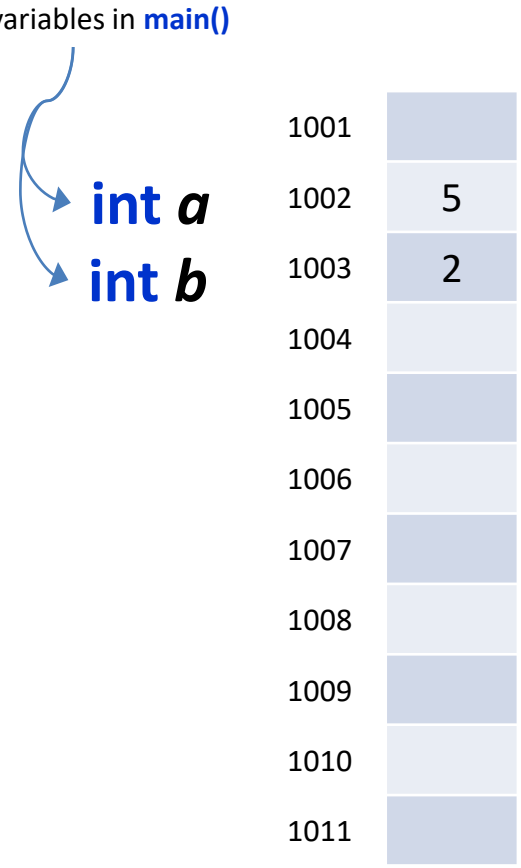
2. Pointers and functions

- **Passing address to the function will have the same effect as pass by reference.**

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

void swap (int * x, int  * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (&a, &b);
    cout << a << " " << b;
    return 0;
}
```

```
5 2
```

Screen output



2. Pointers and functions

- **Passing address to the function will have the same effect as pass by reference.**

Using pointers

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

void swap (int * x, int  * y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (&a, &b);
    cout << a << " " << b;
    return 0;
}
```

```
5 2
```

Screen output

Pass by reference

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

void swap (int &x, int &y){
    int temp = x;
    x = y;
    y = temp;
}

int main(){
    int a = 2, b = 5;
    swap (a, b);
    cout << a << " " << b;
    return 0;
}
```

```
5 2
```

Screen output

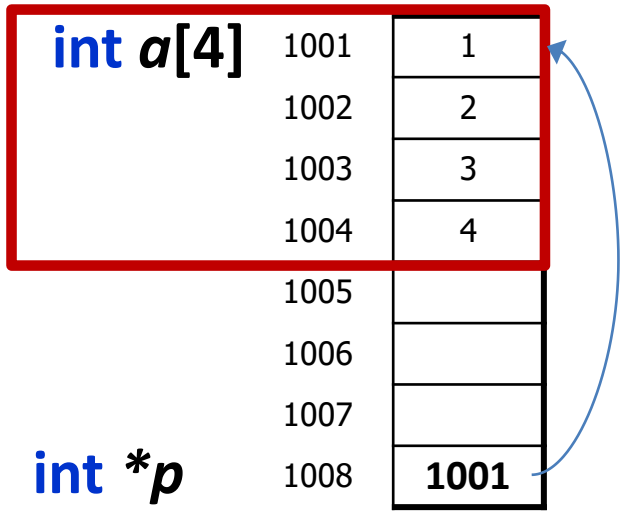
3. Pointers and arrays

- When a pointer is referring to an array, it is storing the address of the 1st slot of the array.
- We can use the increment / decrement operator on pointer variable to go to the next / previous slot of the array.

```
...
int main(){
  int a[4]={1, 2, 3, 4};
  cout << a[0] << a[1] << a[2] << a[3];

  int * p = a;

  cout << *p << *(p+1) << *(p+2) << *(p+3);
  return 0;
}
```

```
1234
1234
```

Screen output



3. Pointers and arrays

- **The following two codes are equivalent.**

p[i] is in fact the short form of *(***p***+***i***)

```
...
int main(){
  int a[4]={1, 2, 3, 4};
  cout << a[0] << a[1] << a[2] << a[3];

  int * p = a;

  cout << *p << *(p+1) << *(p+2) << *(p+3);
  return 0;
}
```

```
...
int main(){
  int a[4]={1, 2, 3, 4};
  cout << a[0] << a[1] << a[2] << a[3];

  int * p = a;

  cout << p[0] << p[1] << p[2] << p[3];
  return 0;
}
```

COMP2113 Programming Technologies / ENGG1340 Computer Programming II
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong