Slot 03 - Pointer Variables

Presenter:

Dr. LE Thanh Tung

Content

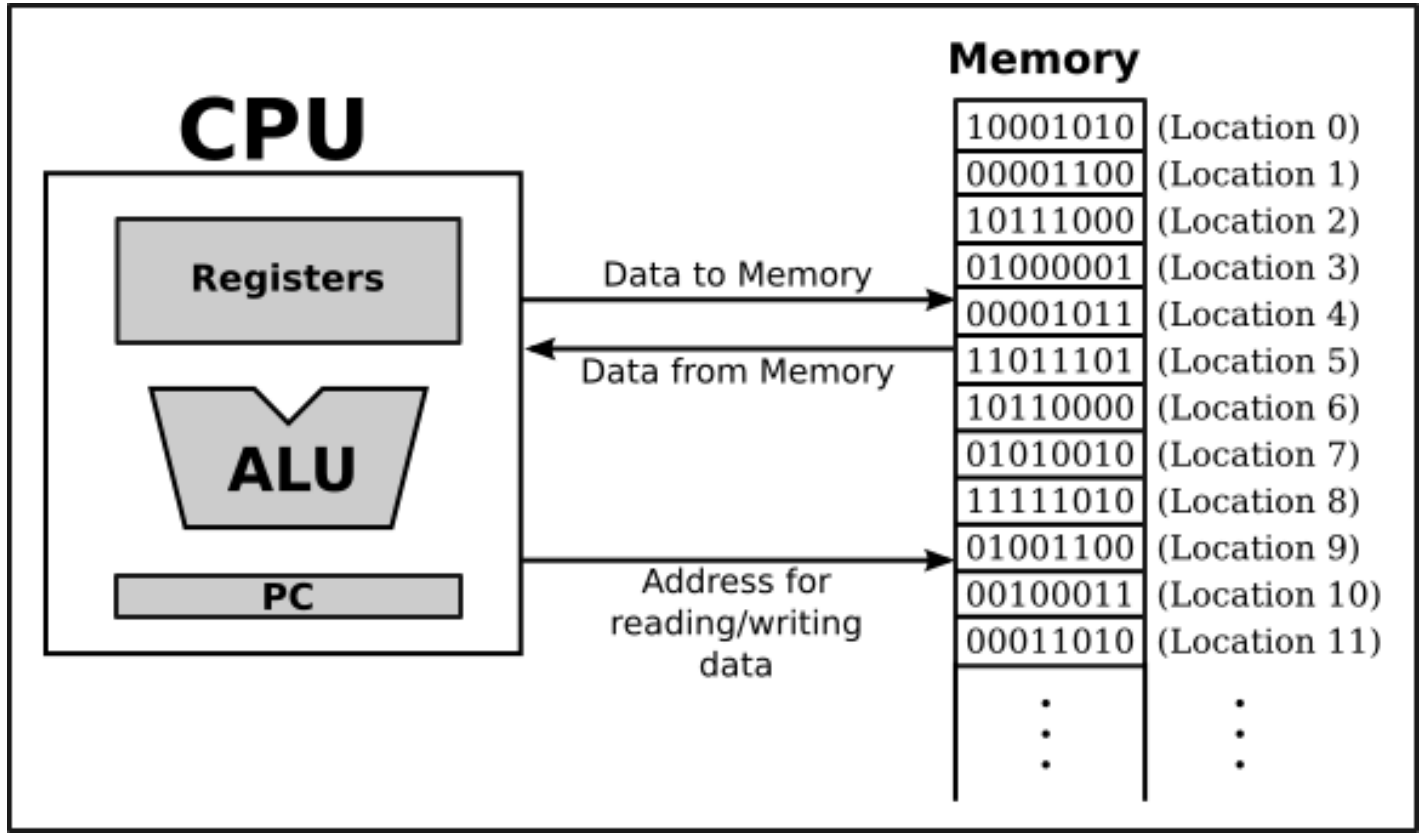
1. Pointer
2. Text File
3. Binary File

Pointer

- Pointer variables are an important concept and one that's traditionally difficult to grasp at first
- Programming with pointer variables in C/C++ is a double-edge sword
  - They can be extremely useful and flexible
  - Misuses of pointers can lead to both bizarre effects and very subtle errors

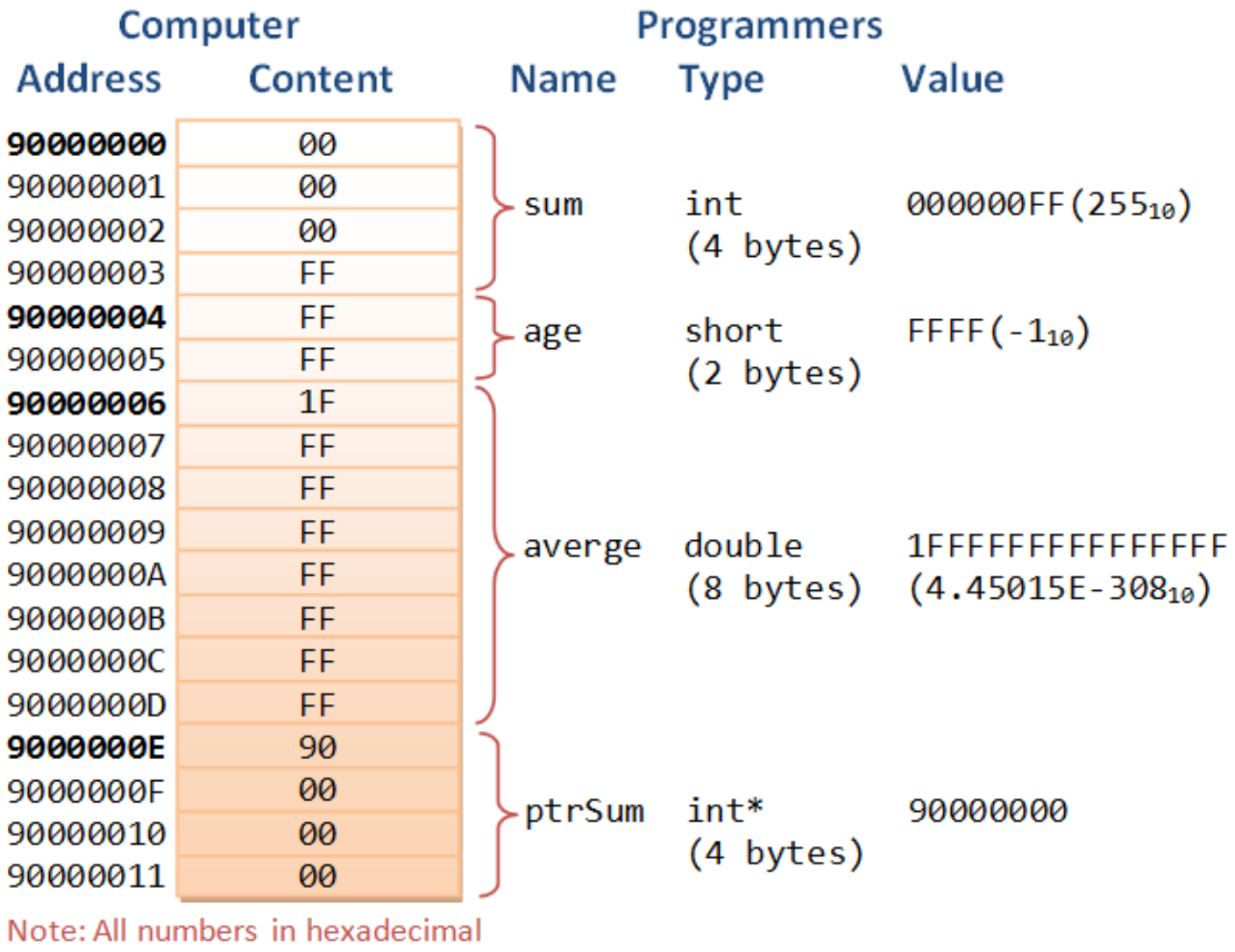
Random Access Memory

- A computer's memory, also known as random access memory (RAM), consists of a sequence of bytes
- Each byte of memory has a unique address
  - Physically, computer addresses start with zero and continue up, one at a time, until they reach the highest address



Variable in C

- The following diagram illustrate the relationship between computers' memory address and content; and variable's name, type and value used by the programmers



Reference Operator

- sizeof(val/type) → the size in bytes of operands
- Operator "&" called the address-of (or address) operator
  - In practice, we do not need to know or use (modify) the physical address of a variable

```
int variable = 123;
...
print("variable value   = %d \n",  variable);
print("variable address = %d \n", &variable);
```

Variable's value

Start with & for variable's address in Main RAM

Pointer

- A pointer variable (pointer) is a special variable, specifically designed to hold a memory address
- Declaration:

```
<typename> * <pointer variable name>;
```

  - E.g: `int* pi`
    - `pi` is a pointer to type `int`
    - `pi` can be used to hold the starting address of the block of memory allocated to a variable of type `int`

Pointer

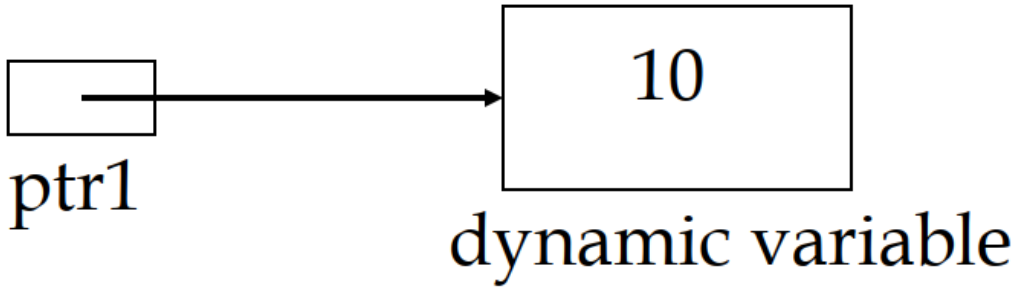
- A pointer variable (pointer) is a special variable, specifically designed to hold a memory address
- Declaration:

```
<typename> * <pointer variable name>;
```

- E.g: `int * pi, n;` `pi = &n;`
  - pi is the pointer to variable n (or pi points to n, or n is pointed to by pi)
  - The value of pi the starting address of the block of memory allocated to variable n
  - The value of pi is the address of variable n

- Pointer diagram visualizes what memory we have allocated and what our pointers are referencing

```
int* ptr1;
ptr1 = new int;
*ptr1 = 10;
...
cout <<*ptr1;   //displays 10
```



```c
#include <stdio.h>

int main() {
  int myAge = 43;  // An int variable
  int* ptr = &myAge;  // A pointer variable, with the name ptr
                      that stores the address of myAge

  // Output the value of myAge (43)
  printf("%d\n", myAge);

  // Output the memory address of myAge
  printf("%p\n", &myAge);

  // Output the memory address of myAge with the pointer
  printf("%p\n", ptr);

  return 0;
```

```
43
0x7ffef71e5404
0x7ffef71e5404
```

Pointer

- A pointer variable is also a variable that requires a block of memory
  - The size of pointers is intimately tied to the computer's processor type, OS, programming language
  - Let's assume that a pointer variable requires 4 bytes of memory

The * operator

- The * operator, called the star operator, is a unary operator that pairs with a pointer variable and turns it into the variable the pointer variable points to
  - It's also called **dereference** pointer operator
  - The & operator and the * operator are inverses of each other

```
int * pi, n = 5;

pi = &n;

n = *pi + 3;

*pi = 10;
```

Pointer Initialization

- Make sure your pointer variables are initialized with valid memory address
  - Using an uninitialized pointer variable is the fastest, and most common, way to crash your program
- If you don't have a value to put in a pointer, set it to the special value `NULL` (or `nullptr` in C++ 11)
  - Now, it's called `null pointer`

Pointers as Function Parameters

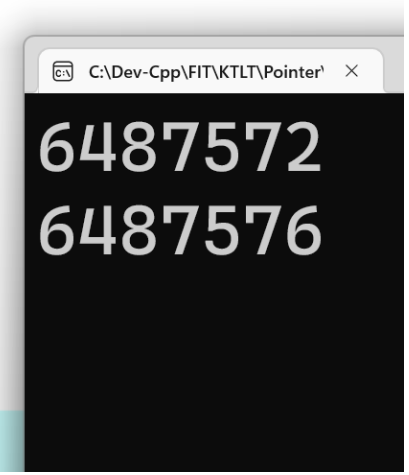
- Pointer variables can be used as function parameters
  - Actual parameters
  - Formal parameters
- The return type of a function can be pointer type
- We can use the pointers as pass-by-reference variables

```
void swap(int* a, int* b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Pointer Arithmetic

- Some arithmetic operations, such as **addition** or **subtraction**, may be performed on pointer variables
- Adding one to a pointer variable increases its value by the size (in byte) of the type to which it points

```
1  #include <stdio.h>
2
3  int main(){
4      int n;
5      int* p = &n;
6      printf("%d\n", p);
7      printf("%d", p + 1);
8      return 0;
9  }
10
```



Constant Pointers & Pointers to Constants

- A pointer to constant points to a constant item
  - The data that the pointer points to cannot change, but the pointer itself can change

```
int i = 5, j = 10;
const int * q;
q = &j;
j++;
*q = 20;     // Error
q = &i;
```

Constant Pointers & Pointers to Constants

fit@hcmus

- A constant pointer is the pointer itself that is constant
  - It must be initialized with a starting value (an address) and it cannot point to anything else

```
int i = 5, j = 10;
int * const p = &i, * q;
q = &j;
p = q;      // Error
p++;        // Error
*p = *q + 5;
```

Constant Pointers & Pointers to Constants

- A constant pointer to constant is a pointer that can neither change its value and nor can change the data it points to

```
int i = 5, j = 10;
const int * const r = &i;
int * q = &j;
i++;
r++;        // Error
*r = 20;    // Error
r = q;      // Error
```

- Try the following code:

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      int a[5] = {1,2,3,4,5};
6      cout << a << endl;
7      cout << &a << endl;
8      cout << &a[0] << endl;
9      return 0;
10 }
```

Array and Pointers

- An array name is a constant pointer pointing to the first element of the array
- We can get the value of each element in array by de-reference operator

```
cout << a[3] << endl;
cout << *(a + 3);
cout << *(&a[0] + 3);
```

Dynamic Memory Allocation

- The size of the problem often cannot be determined at compile time.
- Dynamic memory allocation is the technique of allocating and freeing memory at runtime
- Dynamically allocated memory must be referred to by pointers
- While static memory allocation can only be done on stack memory, dynamic memory allocation can be done on both stack (e.g: recursion) and heap (e.g: allocation)

Dynamic Memory Allocation

- In C programming language:

```
void * malloc(<size>);
```

  - The function allocates a block of <size> bytes of memory and returns a pointer to the beginning of the block
  - If the function failed to allocate the requested block of memory, value NULL is returned

```
free(<pointer>);
```

  - The function deallocates a block of memory previously allocated by a call to malloc, calloc or realloc

Dynamic Memory Allocation

- In C++ programming language:

<pointer var.> = new <type>;

  - The operator new allocates memory of type <type>

<pointer var.> = new <type> [<size>];

  - he operator new is also used to allocate an array of memory of type <type>

delete <pointer>;

delete [ ] <pointer>;

  - The operator delete deallocates a block of memory previously allocated by the new operator

Deallocating

- Once done with dynamic memory
  - we must deallocate it (delete/free)
  - C++ does not require systems to do "garbage collection" at the end of a program's execution
- Example:

```
delete ptr1;
```

  - this does **not** delete the pointer variable
  - Remember assign pointer into **NULL** after deleting the allocating memory,

Example

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory.

And, the pointer ptr holds the address of the first byte in the allocated memory.

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

- Read more: https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

nothrow in new

- If allocation fails, an exception is thrown in this case (default config of new in C++)

```
foo = new int [5];
```

- As using nothrow, the pointer returned by new is a null pointer instead of throwing a bad_alloc exception or terminating the program

```
foo = new (nothrow) int [5];
```

```
1 int * foo;
2 foo = new (nothrow) int [5];
3 if (foo == nullptr) {
4   // error assigning memory. Take measures.
5 }
```

nothrow in new

```
// rememb-o-matic
#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)
    cout << "Error: memory could not be allocated";
  else
  {
    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;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Example

- On the board, let's walk through examples of the following:
  - allocating an array of integers dynamically
  - deallocating that array
  - writing a loop to set the values
  -

THANK YOU
for YOUR ATTENTION