EEET2482/COSC2082

SOFTWARE ENGINEERING DESIGN, ADVANCED PROGRAMMING TECHNIQUES

WEEK 5 – MEMORY ALLOCATION

LECTURER: LINH TRAN

RMIT UNIVERSITY

Review of Pointers

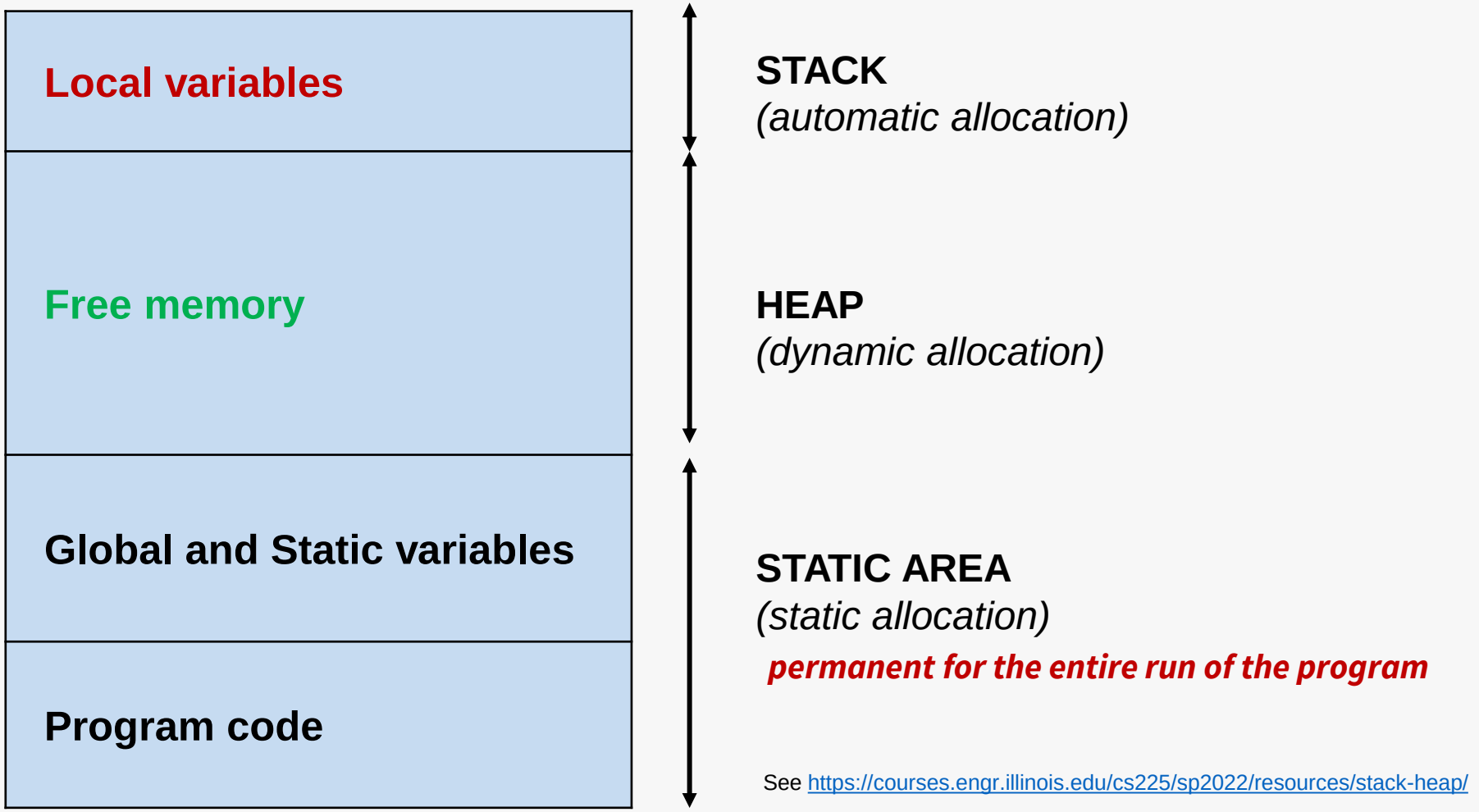
- Example pointer use and memory map:

```
int *p1 = NULL;
int **p2 = NULL;
int x = 19;
int y = 5;

p1 = &x;
p2 = &p1;
*p1 = 542;
y = **p2 + 5;
```

Memory Address	Variable Name	Contents
0x100	p1	0
0x101	p2	0
0x102	x	19
0x103	y	5

Memory Address	Variable Name	Contents
0x100	p1	0x102
0x101	p2	0x100
0x102	x	542
0x103	y	547

Memory Allocation



See https://courses.engr.illinois.edu/cs225/sp2022/resources/stack-heap/

STACK

- Store local variables.
- It's a **LIFO** (Last-In - First-Out) structure. *New local variables are pushed onto the stack upon being created, and are freed up from the stack automatically on upon being discarded.*
- Stack overflow: when all the memory in the stack has been allocated, and further allocations begin overflowing into other sections of memory.

Note: A common mistake is to return a pointer to a local variable of a function --> may cause to program to crash !!!

HEAP

- Free memory region (usually shared by all programs) that can be allocated explicitly by programmers.

- You can *control the exact size and lifetime* of these memory allocations to achieve better efficiency.

  You must free these memory locations after usage, otherwise, you may run into ***memory leaks*** *which may cause your program to crash or cannot allocate memory later.*

Dynamic memory allocation with HEAP

- Request allocation in heap by **new** operator:

```
pointer = new data_type;
pointer = new data_type (initial_value);
pointer = new data_type [size_of_array];
```

```
int *a = new int(25);
float *b = new float(75.25);
int *arr_p = new int[10];
```

Note: *If there is not enough memory in the heap to allocate, it may cause an exception error and return a NULL pointer.*

- Free up memory allocated by **delete** operator:

```
delete allocated_pointer;        //free up a single item
delete[] allocated_pointer;      //free up an array
```

```
delete a;
delete b;
delete arr_p;
```

NOTE: *We can allocate and deallocate OBJECTS using **new** and **delete** keywords as normal data type*

Example

```
#include <iostream>

class Student {
public:
    int score;
};

int main() {
    int size;
    std::cout << "Enter total number of students: ";
    std::cin >> size;

    //memory allocation for an array
    Student* ptr = new Student[size];

    //store and read values
    std::cout << "Enter scores of students:" << "\n";
    for (int i = 0; i < size; ++i) {
        std::cout << "Student" << i + 1 << ": ";

        std::cin >> ptr[i].score; //store values
        std::cout << "> Stored: " << ptr[i].score << "\n"; //read values
    }

    //free up memory location (allocated for ptr before)
    delete[] ptr;

    return 0;
}
```

Class Deconstructor

- A destructor is special member function of a class, which will be automatically called before an object is destroyed.
- Destructors have ***same name with classname preceded by ~***, no parameters and no return type. *By default, the compiler creates a default destructor for each class.*
- Usage: *if we do dynamic memory allocations in a class, we can write a user-defined destructor to release memory before the class instance is destroyed*. This must be done to avoid memory leak.

  See https://www.geeksforgeeks.org/destructors-c/

Example

```
#include <iostream>

class Data {
private:
    std::string name;
    int* arr;

public:
    Data(std::string name) { // constructor
        //std::cout << "Constructor of " << name << " is called \n";
        this->name = name;
        arr = new int[1000]; //dynamic memory allocation
    };

    ~Data() { // destructor
        //std::cout << "Destructor of " << name << " is called \n\n";
        delete[] arr; //remove this one will cause Memory Leak (computer hang)
    };
};
```

```
//A function with local object to test
void myFunc(int i){
    std::string name = "data" +
std::to_string(i);
    Data myData(name); //create a local object
}


int main() {
    //call the function many times
    for (int i = 0; i < 10000000000; i++) {
        myFunc(i);
    }

    return 0;
}
```

When to use dynamic memory allocation?

1. **When you need a lot of memory**

   The size of the stack memory per process is usually between 1 to 8 MB, while *the size of the heap memory is only limited by the available physical memory*

2. **When you need the data after the function returns**

   Stack memory gets destroyed when the function ends but *heap memory is only freed when you want*

3. **When you are building a data structure (e.g. an array of structure) of unknown size at compile time which could later become big**

   Dynamic memory allocation allows you to *request an exact amount of memory when you need at runtime*

Check for Error of Memory allocation

Dynamic allocation may fail if there is not enough memory space. Two ways to check:

1. C++ will throw a **`std::bad_alloc`** exception. Program will quit if exception is not handled in your code

   --> Use **`try-catch`** statement to check and handle the exception error.

2. Specifically specify **`(std::nothrow)`** to ignore exception and check for NULL pointer

```
type *newPointer = new (std::nothrow) type;
if (newPointer == NULL) {
   // Error, dynamic memory not allocated successfully
}
```

Exceptions: special events thrown when an error or unexpected circumstance happens in program execution.

`try-catch` statement

- **`try`** block: contains all program code that you want to catch exceptions.
- **`catch`** block: catches exception(s) thrown for code in try block
- ***A `try` block must be followed by at least one `catch` block*** (*can have multiple overloading catch blocks for each exception or generic catch statement*). Program code continues execution after catch block.
- Besides system defined exceptions, we can use **`throw`** keyword to throw a custom exception.

Example

```
#include <iostream>

int main() {

    try { // all code you want to handle possible exceptions

        //request memory allocation (very large size will throw bad_alloc exception)
        int *ip = new int[10000];

        int age;
        std::cout << "Enter age: ";  std::cin >> age;
        if (age <= 16) {
            throw 101; //throw a custom error code (here is an integer value)
        }
    }

    catch (std::bad_alloc& ba) { //Handles std::bad_alloc exception
        std::cerr << "bad_alloc exception caught: " << ba.what() << '\n';
    }

    catch (int errorCode) { // Handles custom exception
        std::cerr << "Access denied (16+). Error Code: " << errorCode;
    }

    catch (...) { // Handles all other generic exeptions
        std::cerr << "Generic Exceptions !\n";
    }

    return 0;
}
```

Range-based for loop

- Syntax:

```
for ( data_type range_var : range_expression ){
    statement(s);
}
```

See: https://docs.microsoft.com/en-us/cpp/cpp/range-based-for-statement-cpp?view=msvc-170

```
// Iterate through a braced list (by value)
for (int i : {0, 1, 2, 3, 4, 5}) {
    std::cout << i << ' ';
}
std::cout << '\n';

// Iterate through the array (by value).
int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for (int x : arr) {  // Access by value
    std::cout << x << " ";
}
std::cout << "\n";
```

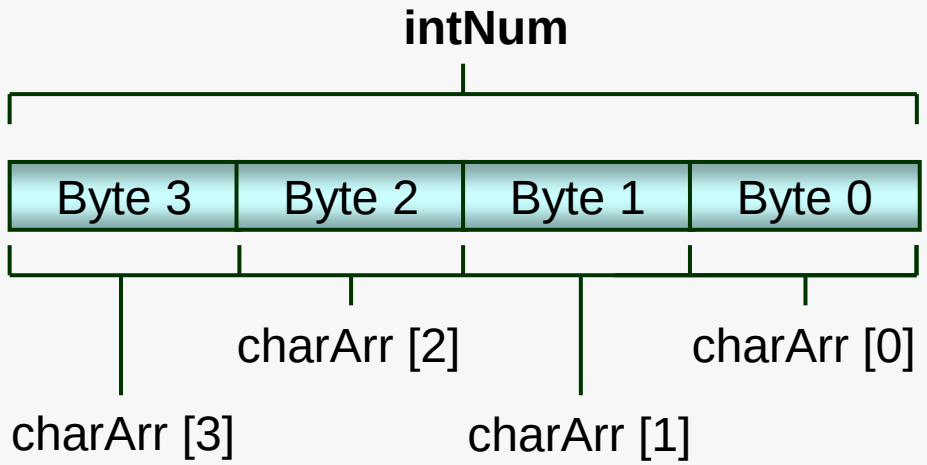
```
// Iterate through a string (by value).
// "auto" keyword: type is automatically
// inferred from the range
std::string str1 = "Hello!";
for (auto ch : str1) {
    std::cout << ch << ' ';
}
std::cout << "\n";


// Iterate through a string (by reference).
// Use when modify is needed
std::string str2 = "Hello!";
for (auto &ch : str2) {
    if (ch == 'l') {
        ch = 'L';
    }
}
std::cout << str2 << "\n";
```

Union

- A *union* is a collection of variables of different data types that **share memory location** *(get allocated with the largest size among member variables).*
- In C++, like *struct*, *union* also define a class (all members are public by default).
- Since *union* is also originated from C, in practice, people usually only use union in a C-style maner **(i.e. with attributes, but no member methods).**

Example

```
union u_type {
  int  intNum;
  char charArr[4];
};
```




```
#include <iostream>

union u_type {
  int  intNum;
  char charArr[4];
};

int main() {
    u_type myUnion;
    std::cout << "Size = " << sizeof(u_type) << "\n";

    //Access intNum elements
    myUnion.intNum = 0x00434241; //0x41-43 is Ascii values of 'A'-'D'

    //Access charArr elements
    std::cout << myUnion.charArr << "\n";
    std::cout << myUnion.charArr[0] << "\n";
    std::cout << myUnion.charArr[1] << "\n";
    std::cout << myUnion.charArr[2] << "\n";
    std::cout << myUnion.charArr[3] << "\n";

    return 0;
}
```

```
Size = 4
ABC
A
B
C
```