Lecture 11

- C-strings continued
  - C++ I/O
  - Command line arguments
- Dynamic memory allocation

C-String + C++ I/O

```
char a[80], b[80];

cin >> a >> b; cout << a << " - " << b;

Input: To be or not to be
Output: To - be
```

- Output using << operator
  - E.g. **char s[] = "hello"; cout << s;**
- Input using >> also possible, **BUT**
  - leading whitespaces (blanks, tabs, newline) are skipped
  - reading stops at next whitespace
  - string length in input may be larger than string variable! **Unsafe! SO DON'T USE >> ON C-STRINGS!**

Better Solution

- Input stream function
  - **void getline(char s[], int max_total_len);**
- Reads entire input line into string s including whitespaces
- Copies up to max_total_len-1 characters
- End-of-line character ('\n') is not copied
- **Even better:** C++ string class (later)

```
char a[5];

cin.getline(a, 5); cout << a;

Input:  "123456789\n"
Output: "1234"
```


Command-Line Arguments

```
// print all command-line options
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{ for (int i=0; i < argc; ++i)
    cout << "arg-" << i
         << " " << argv[i]
         << endl;
}
```

```
./foo -o 1 2 3

output:

arg-0 ./foo
arg-1 -o
arg-2 1
arg-3 2
arg-4 3
```

- main prototype: int main(int argc, char *argv[]);
  - argc: number of command-line arguments +1
  - argv: array of pointers to command-line args.
  - argv[0]: pointer to program name
  - argv[1]: pointer to first argument, ...

Dynamic Memory Allocation Operators new and delete

- Local variables and functions parameters are located on the **stack** (LIFO data structure)
- Dynamic memory is allocated from a different part of memory called **heap**
- Operator **new** dynamically allocates memory
- Operator **delete** is used to release it when no longer needed – can be done later, even in a different func.
- As always, **YOU** are in control because the compiler cannot know when memory is no longer needed and can be deleted.
- C/C++ does not have a garbage collector

Operator new

```
int *p = new int;   // allocates space
                    // for an int
                    // p now points to it

if (!p) { cerr << "out of memory" << endl;
          exit(-1); }

*p = 0;             // use allocated memory
```

- Syntax: **new** <type>
- Allocates space for a variable of type <type> on the **heap** and returns a pointer to it
- No initialization if <type> is a basic C type
- If no memory is available new returns 0

Operator delete

```
int *p = new int;
...
// free memory when *p is no longer used
delete p;
p = 0;  // safeguard
```

- Frees memory when it is no longer used
- Syntax: **delete** <pointer-to-allocated-mem>
- Good practice: **set pointer to 0** after delete to prevent further access of this address
- 0 special pointer value: can be assigned to any pointer variable regardless of type
- 0 not part of process memory. Can indicate no memory, invalid pointer, no successor, etc.

Dynamic Arrays

```
float *p = new float[100];
if (!p) { exit(-1); } // out of memory
...
for (int i=0; i < 100; ++i) p[i] = 0.0;
...
// free memory when *p is no longer used
delete [] p;
p = 0;  // safeguard
```

- Syntax: **new** <type>**[**<num-of-elements>**]**
- Allocates an array of elements of type <type>
- Elements are not initialized for basic C types
- When no longer used free memory with **delete [ ]** <pointer-to-dynamic-array>

Good new/delete Practice

- new/delete come in pairs: for every new there should be a delete in your program
- More specifically:
  - for every **new** at least one corresponding **delete**
  - for every **new[]** at least one corresponding **delete[]**
- **Helps avoiding memory leaks**

Speed / Memory Issues

- Allocating dynamic memory is **SLOW**
- Program has to maintain list of available memory blocks
- If speed is important try to minimize news/deletes E.g. by **reusing arrays**
- new allocates **more memory** than you think (overhead usually 4 or 8 bytes per call, getting better)
- Allocating arrays is therefore more efficient than single variables
- You can roll your own memory allocation by overloading the new operator (later)

Memory Allocation in C

- There are no new/delete operators in C
- Use library function calls
  - malloc : allocates memory
  - free : releases memory
- To learn about them: man malloc

```
float *p = (float*) malloc(100*sizeof(float));
if (!p) { exit(-1); } // out of memory
...
for (int i=0; i < 100; ++i) p[i] = 0.0;
...
// free memory when *p is no longer used
free(p);
p = 0;  // safeguard
```