HIGH-LEVEL PROGRAMMING 2

Dynamic Memory in C++ by Prasanna Ghali

Heap Memory Allocation & Deallocation Functions: <cstdlib>

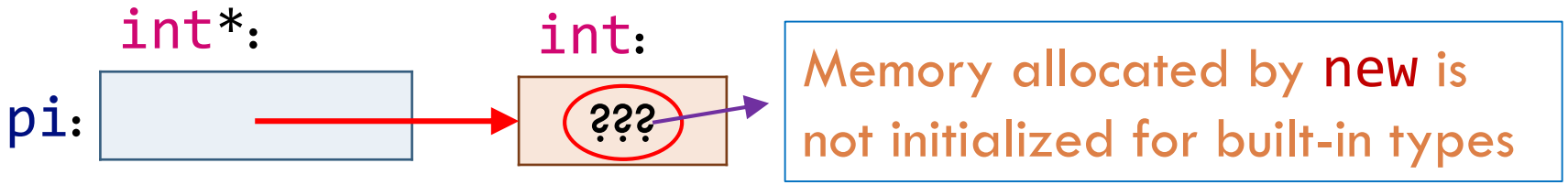

```
// declared in <cstdlib>
namespace std {

// functions for dynamically allocating heap memory
void* malloc(size_t size);
void* calloc(size_t count, size_t size);
void* realloc(void *ptr, size_t size);

// function for returning dynamically allocated
// memory back to heap
void free(void *ptr);

}
```

Program Memory Layout

Free Store Allocation

- If T is type of object, *new expression* new T allocates object of type T on free store
  - This first form of new operator returns pointer to memory allocated for single object
  - Pointer value is address of first byte of the memory
  - Pointer points to an object of specified type
  - Pointer *doesn't know* how many elements it points to

```
int *pi {new int};
```



Initialization of Allocated Memory

- Memory allocated by new operator is not initialized for built-in types
- You can change that using { } for initialization

```
int *pi    {new int{11}};   // ok: modern C++
int *pi2   {new int(12.1)}; // ok: old-style C++
int *pi3   {new int()};     // ok: initialize to 0
int *pi4   {new int{12.1}}; // error
double *pd {new double {11.1}}; // ok
```

The Null Pointer

- If you've no other pointer to use for initializing a pointer, use null pointer `nullptr`
- Name `nullptr` for null pointer is new in C++11 – in older code, people often use `0` (zero) or `NULL`

```
double *pd {nullptr} // the null pointer
// some code here ...
if (pd != nullptr)   // consider pd valid

// even shorter ...
if (pd) // consider pd valid
```

Free Store Deallocation (1/2)

- For large programs and for long-running programs, freeing of memory for reuse is essential!!!

```
// leaks memory
double foo(int res_size) {
  double acc{};

  for (int i{}; i < res_size; ++i) {
    double *p { new double{} };
    // use p to calculate results to be put in res
    acc += *p;
  }
  return acc;
}
```

Free Store Deallocation (2/2)

- If p is pointer variable with value from *new expression*, *delete expression* `delete p` returns memory to free store

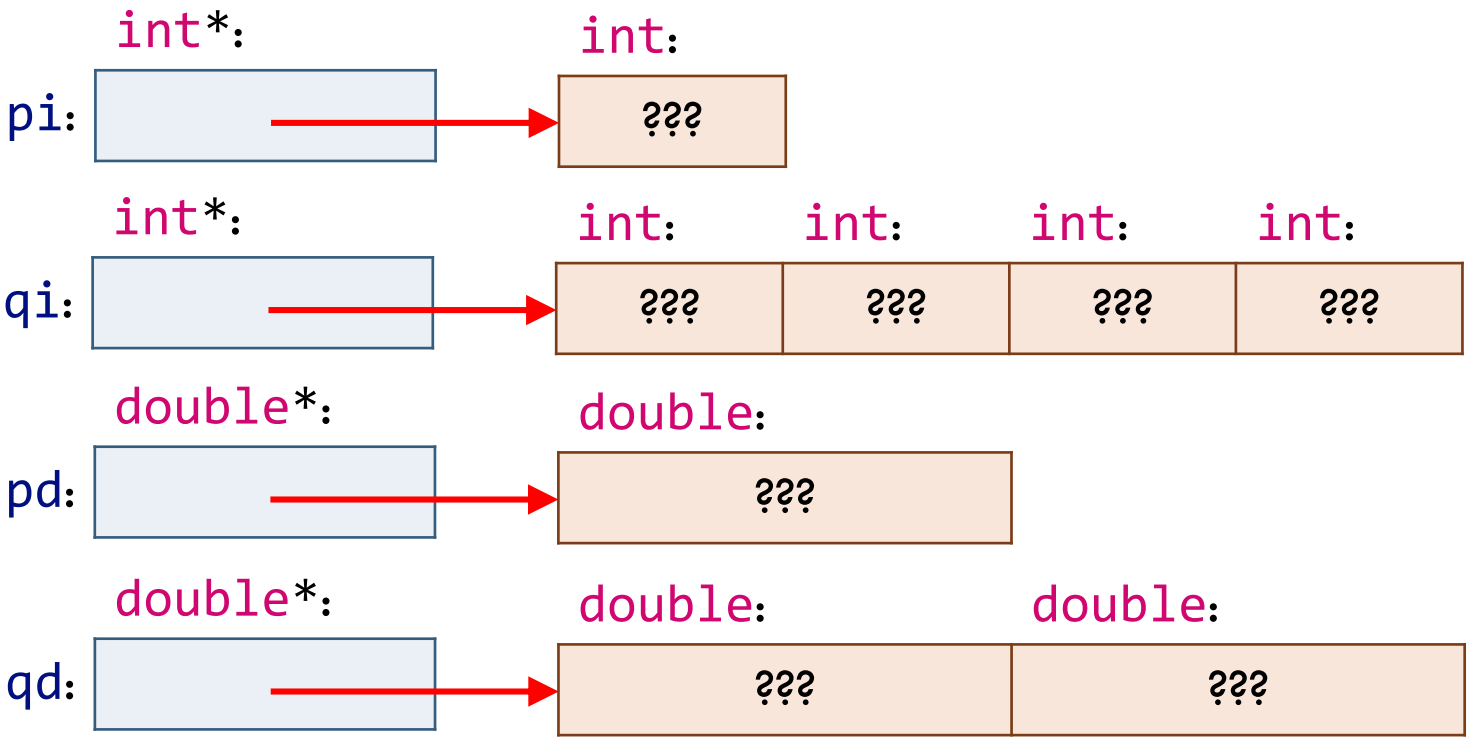
```
// doesn't leak memory anymore...
double foo(int res_size) {
  double acc{};

  for (int i{}; i < res_size; ++i) {
    double *p { new double{} };
    // use p to calculate results to be put in res
    acc += *p;
    delete p;
  }
  return acc;
}
```

Free Store Allocation of Dynamic Array (1/2)

- If T is a type and n is non-negative integral value, *array new expression* new T[n] allocates array of n objects of type T on free store
  - This 2nd form of new operator returns pointer to allocated memory
  - Pointer value is address of first byte of the memory
  - Pointer points to first element of specified type
  - Pointer *doesn't know* how many elements it points to

Free Store Allocation of Dynamic Array (2/2)


```
int *pi    { new int };        // allocate one int
int *qi    { new int[4] };     // allocate array of 4 ints
double *pd { new double };     // allocate one double
double *qd { new double[2] }; // allocate array of 2 doubles
```



Initialization of Allocated Memory

- We can specify initializer list to initialize elements of dynamic array allocated by *array new expression*

```
int *pai1 { new int [5] {1,2,3,4,5} };
int *pai2 { new int []  {11,22,33,44,55} };

double *pad3 { new double [4] {1, 2, 3, 4} };
double *pad4 { new double [3] {1.1, 2.2, 3.3} };
```

Doesn't compile in g++ but compiles in clang++ and cl [Microsoft]

Free Store Deallocation (2/2)

- If p is pointer variable with value from *new expression*, *delete expression* `delete p` returns memory to free store

```
// doesn't leak memory anymore...
double foo(int res_size) {
  double acc{};

  for (int i{}; i < res_size; ++i) {
    double *p { new double{} };
    // use p to calculate results to be put in res
    acc += *p;
    delete p;
  }
  return acc;
}
```

Free Store Deallocation

- If p is pointer variable with value from *array new expression*, *array delete expression* `delete[] p` returns memory to free store

```
double* calc(int res_size, int max) {
  double *p { new double[max] };
  double *res { new double[res_size] }
  // use p to calculate results to be put in res
  delete[] p;
  // caller responsible for memory allocated for res
  return res;
}

double *r { calc(100, 1000) };
// use r ...
delete[] r; // don't need memory anymore: free it
```

Memory Exhaustion

- Beware!!! Unlike `malloc`, `new` and `new[]` don't return `nullptr` when free store memory is exhausted!!!
- Instead, they throw `std::bad_alloc` exception [exceptions are covered in 2nd half of semester]
- See *exhaust.cpp* where check for `nullptr` fails when free store is exhausted
- Since exiting or aborting our program is only option when free store is exhausted, it doesn't much matter!!!

Caveat: Don't Use C Standard Library!!!

- Unlike `malloc` and `free`, `new` and `new[]` know about constructors while `delete` and `delete[]` know about destructors
- `malloc` just allocates memory while `new` allocates memory and *then calls appropriate constructor to initialize allocated object*
- `free` just deallocates memory while `delete` *calls destructor* and then deallocates memory
- Recall that built-in types don't have ctors and dtors
- See *num.hpp, num.cpp, num-driver.cpp*

Caveat: Don't Mix C and C++ Concepts!!!

- Don't use `free` on pointers that point to memory returned by `new` or `new[]`
- Don't use `delete` or `delete[]` on pointers that point to memory returned by `malloc`

Caveat: Don't Mix Different Forms Of `new` And `delete`

- Two forms of `new`
  - `new p` allocates memory for individual object
  - `new[] p` allocates memory for array of objects
- Two forms of `delete`
  - `delete p` frees memory for individual object allocated by `new`
  - `delete[] p` frees memory for array of objects allocated by `new[]`
- *It is programmer's tedious job to use right version*

Pointers Are Error Prone

- Dereferencing uninitialized pointers
- Dereferencing `nullptr`s
- Reading uninitialized objects that are dynamically allocated
- Failing to `delete` [or `delete[]`] allocated memory causing memory leak
- Calling `delete` rather than `delete[]` and vice versa
- Accessing `delete`d memory
- Double `delete`ing dynamically allocated objects
- Premature deletion causes dangling pointers
- Off-by-one array subscripting

Pointers Are Error Prone

- Use Valgrind to debug memory bugs!!!
- Read handout for detailed explanations!!!