Array Limitations in C++

Standard C++ does not let you do this*:

```cpp
int sz;
cout << "What size do you need?" << endl;
cin >> sz;
int arr[sz]; // compiler error
...```

*Strangely, later versions of C do allow this. Confusingly, so does g++.

Dynamic Array Allocation

So what if you know you’ll need an array, but not the size (at compile time)?

```cpp
int sz;
cout << "What size do you need?" << endl;
cin >> sz;
int *arr = new int[sz];
...```

Where Does Memory Come From?

The stack: local variables, function arguments, return values. Grows "down".

The heap: dynamically allocated memory. Grows "up".

Global and static variables, constants.

Program code. Read only!

The Stack

- Holds “stack frames” aka “activation records”
- Each function call results in a new stack frame
- Each stack frame contains memory for:
  - Local variables declared in the function
  - Parameters passed into function
  - Return address for function
- When the function is exited, all of this memory is returned to the stack automatically.

Data Segment/BSS

Global and static variables:
- Only ever one instance of them
- Get stored in their own special area
- Memory is pre-allocated, fixed in size
The Heap

A big ol’ hunk of memory!

- Get pieces of it ("allocate memory") using new
- Pieces stay allocated until explicitly released by use of delete

Heap memory has a lifetime independent of scope – it can be passed out of functions, for instance. You can’t do that with local variables!

Stack vs Heap

- Stack: local aka automatic variables and arrays:
  - int z;
  - foo f;
  - double darray[100];
  - Memory for these is allocated on the stack when they come into scope, is returned to the stack when they go out of scope (e.g., when function returns).

- Heap: dynamically allocated objects and arrays:
  - int* p = new int;
  - foo* fp = new foo;
  - double* dptr = new double[100];
  - All of these live on the heap. They will exist until explicitly deallocated by user code.

Dynamic Arrays

Allocate dynamic arrays using new:

```cpp
double *darray = new double[1024];
```

Use the array pointer just like a regular array:

```cpp
for (int j = 0; j < 1024; j++)
    darray[j] = j;
```

Always clean up (deallocate) when you are done:

```cpp
delete[] darray;
```

Dynamic Arrays: Rules

Never:

- Dereference a pointer which has not been set to valid memory (using new or &)
- Dereference a pointer to memory which has been deallocated (a dangling pointer)
- Change or lose a pointer which is pointing to dynamically allocated memory (or you won’t be able to deallocate – this causes a memory leak)
- Use delete on a pointer which isn’t pointing to dynamically allocated memory (e.g., a dangling or NULL pointer)

Pointers, Objects, and Dynamic Memory

Consider this simple class:

```cpp
class student {
    public:
        string name;
        student() { ; }
        student(string n) { name = s; }
        void eat();
        void sleep();
};
```

Creating New Objects: Stack

If we want to create a student locally:

```cpp
student student1;
student student2("Kirk");
```

- These are created on the stack.
- They will vanish when exiting the current scope.

student1 is created using the default constructor:

```cpp
student();
```

student2 is created using another constructor:

```cpp
student(string s);
```
Creating New Objects: Heap

We can also create single objects \textit{dynamically}:

\begin{verbatim}
Pointers: 
student* sp1 = new student; 
student* sp2 = new student("Picard"); 
\end{verbatim}

These are created on the \textit{heap}.
They will live forever unless deleted:
\begin{verbatim}
delete sp1; 
delete sp2; 
\end{verbatim}

Note, again, the two different constructors.

Working With Object Variables

Consider:
\begin{verbatim}
student* p = new student; 
\end{verbatim}

We know that we can do:
\begin{verbatim}
student1.name = "Sisko"; 
student1.eat(); 
\end{verbatim}

What can we do with \textit{p}?

Working with Object Pointers

We have:
\begin{verbatim}
student* p = new student; 
\end{verbatim}

We could just dereference (perfectly fine!)
\begin{verbatim}
(*p).name = "Janeway"; 
(*p).sleep(); 
\end{verbatim}

C++ gives us another operator we can use directly:
\begin{verbatim}
p->name = "Archer"; 
p->sleep(); 
\end{verbatim}

The Destructor

The counterpart to the constructor:
\begin{itemize}
  \item No return type 
  \item Name is ~ followed by class name, e.g., 
        ~student();
  \item Never takes a parameter!
\end{itemize}

The destructor is called automatically when:
\begin{itemize}
  \item A local (stack allocated) object goes out of scope 
  \item delete is called on a dynamically allocated object 
\end{itemize}

Arrays of Objects

We can also use new to create arrays of objects:
\begin{verbatim}
int n = 100; 
student* arr = new student[n]; 
\end{verbatim}

The \textit{default constructor} is used to create every object in the array.

Note different syntax if we use \textit{pointers} vs array indexing:
\begin{verbatim}
for (int i = 0; i < n; i++) 
  arr[i].gpa = 4.0; 
\end{verbatim}

\begin{verbatim}
for (student* p = arr; p < arr + n; p++) 
  p->gpa = 4.0; 
\end{verbatim}

As with base types, we use delete[] on dynamically allocated arrays of objects:
\begin{verbatim}
delete[] arr; 
The destructor is called on every object in the array. 
\end{verbatim}

2-D Arrays

For multi-dimensional arrays, you need arrays of pointers, e.g.:
\begin{verbatim}
// make a 5x7 2-D int array 
int **arr; 
arr = new int*[5]; // allocate array of int pointers 
for (int i = 0; i < 5; i++) 
  arr[i] = new int[7]; // allocate columns 
arr[2][6] = 42; // etc.
\end{verbatim}

// clean up 
for (int i = 0; i < 5; i++) 
delete[] arr[i]; // deallocate columns 
delete[] arr; // deallocate array of int pointers
Up Next

- Friday, Sept. 29
  - Lab 6 – Memory
- Monday, Oct. 2
  - Midterm Review
  - Project 2 Due
  - Lab 6 Due
- Wednesday, Oct. 4
  - Midterm 1 (in class)