CSCI 2270: Data Structures

Lecture 05: C++ Review: Pointers, References, Parameter-Passing, and Best practices

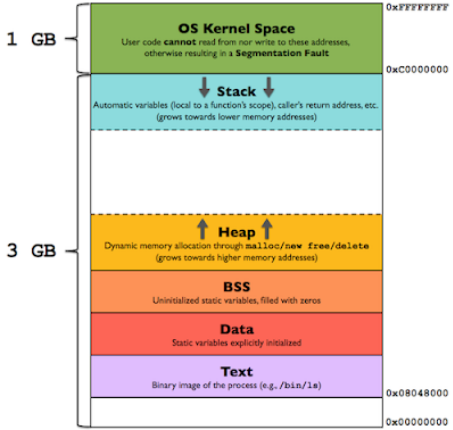
Ashutosh Trivedi

Department of Computer Science
UNIVERSITY OF COLORADO BOULDER

Stack and Heap

Pointers and References

In-Memory Layout of a Program



Stack Vs Heap: Pros and Cons

Stack:

- very fast access
- don't have to explicitly free variables
- space is managed efficiently by CPU,
- memory will not become fragmented
- local variables only
- limit on stack size (OS-dependent)
- variables cannot be resized

Heap

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- you must manage memory (you're in charge of allocating and freeing variables)

When to use the Heap?

– If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.

When to use the Heap?

- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.
- If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack.

When to use the Heap?

- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.
- If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack.
- If you need variables like arrays and structs that can change size dynamically (e.g. arrays that can grow or shrink as needed) then you will likely need to allocate them on the heap, and use dynamic memory allocation functions to manage that memory "by hand".

Stack Memory is limited!

- A large array on the stack frame:

```
// program25.cpp
#include<iostream>
int main(int argc, char* argv[])  {
  int big[1000000]; // Trying to allocate a huge array on stack
  // Since stack memory is limit (see ulimit -a), it will result in segfault
  return 0;
}
```

Stack Memory is limited!

- A large array on the stack frame:

```
// program25.cpp
#include<iostream>
int main(int argc, char* argv[])  {
  int big[1000000]; // Trying to allocate a huge array on stack
  // Since stack memory is limit (see ulimit -a), it will result in segfault
  return 0;
}
```

- A deep stack:

```
// program26.cpp
#include<iostream>
void foo(int num);
int main(int argc, char* argv[])  {
  if (argc==2) foo(atoi(argv[1]));
  return 0;
}
void foo(int num)
{
  if (num > 0) foo(num-1);
}
```

Heap Memory

- A large array on the heap:

```
// program27.cpp
#include<iostream>
int main(int argc, char* argv[])  {
  int *big = new int[10000000]; // Trying to allocate a huge array on heap
  // No problem what-so-ever!
  return 0;
}
```

Heap Memory

- A large array on the heap:

```
// program28.cpp
#include<iostream>
void untidiness(int depth);
int main(int argc, char* argv[])  {
  if (argc==2) untidiness(atoi(argv[1]));
  return 0;
}
void untidiness(int depth) {
 int *big = new int[10000000]; // Trying to allocate a huge array on heap
 // Use this arrary
 if (depth > 0) untidiness(depth-1);
}
```


DO YOU HAVE AN ITEM IN YOUR HEAP MEMORY THAT DOESN'T SPARK JOY?
THANK IT FOR ITS SERVICE AND GET RID OF IT USING "DELETE"
makeameme.org

Heap Memory

- A large array on the heap:

```
// program29.cpp
#include<iostream>
void untidiness(int depth);
int main(int argc, char* argv[])  {
  if (argc==2) untidiness(atoi(argv[1]));
  return 0;
}
void untidiness(int depth) {
   int *big = new int[10000000]; // Trying to allocate a huge array on heap
   // Use this array
   delete[] big;
   big = 0;
   if (depth > 0) untidiness(depth-1);
}
```

Stack and Heap

Pointers and References

Pointers

- Pointer variables hold addresses of other variables.

```
int *pi; // pointer to int
char **ppc; // pointer to pointer to chars
int *ap[15]; // array of pointer to int
int (*fp)(char *); // function pointer
int * f(char *); // return value a pointer
```

Pointers

- Pointer variables hold addresses of other variables.

```
int *pi; // pointer to int
char **ppc; // pointer to pointer to chars
int *ap[15]; // array of pointer to int
int (*fp)(char *); // function pointer
int * f(char *); // return value a pointer
```

- Address-of (&) and de-referencing (*) operators:

```
char c = 'a'
char *p = &c;
char c2 = *p
```

Pointers

- Pointer variables hold addresses of other variables.

```
int *pi; // pointer to int
char **ppc; // pointer to pointer to chars
int *ap[15]; // array of pointer to int
int (*fp)(char *); // function pointer
int * f(char *); // return value a pointer
```

- Address-of (&) and de-referencing (*) operators:

```
char c = 'a'
char *p = &c;
char c2 = *p
```

- No object is addressed at 0 and hence a pointer equals to 0 means not pointing to any object.

Pointers

```
// program30.cpp
#include<iostream>
int main(int argc, char* argv[])  {
  char c = 'a';
  char *cp = 0;

  cp = &c;
  std::cout << *cp << std::endl;

  *cp = 'b';
  std::cout << c << std::endl;
  std::cout << *cp << std::endl;

  cp = &*&*&c;
  std::cout << *cp << std::endl;

  return 0;
}
```

Best practice 1.

– *Set pointer to 0 immediately after deleting the memory pointed by it.*

```
char * cp = new char('c'); // Heap allocation
std::cout << *cp << std::endl;

delete cp; // Heap deletion
cp = 0; // set to zero immediately after freeing.

delete cp;
// Why? reading
// std::cout << *cp << std::endl;
```

Arrays

- An array is a collection of elements of the same type.
- Given a variable of type T, and array of type $T[N]$ holds an array of N elements, each of type N.
- Each element of the array can be referenced by its index that is a number of 0 to $N-1$.

```
// program8.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int ia[3]; //Array of 3 ints with garbage values
  std::cout << ia[1] << std::endl;
  float fa[] = {1, 2, 3}; //Array of 3 floats initialzed: size automatically computed
  std::cout << fa[2] << std::endl; // Read different values
  return 0;
}
```

Arrays (Statically Declared Arrays)

```
// program8.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int ia[3]; //Array of 3 ints with garbage values
  std::cout << ia[1] << std::endl;
  float fa[] = {1, 2, 3}; //Array of 3 floats initialzed: size automatically computed
  std::cout << fa[2] << std::endl; // Read different values
  return 0;
}
```

1. Static Array storage is contiguous.
2. Array bound must be a constant expression. If you need variable bounds, use a `vector`.
3. What happens when initialization and array size mismatch?
4. Multi-dimensional arrays (contiguous in row-order fashion!).

Arrays (Dynamically Declared Arrays)

```
// program9.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int* pa = 0; // pa is a pointer to integers
  int n;
  std::cout <<  "Enter dynamically allocated array size:";
  std::cin >> n;
  pa = new int[n];
  for (int i = 0; i < n; i++) {
    pa[i] = i;
  }
  // Use a as a normal array
  delete[] pa; // When done, free memory pointed to by a.
  pa = 0; //// Clear a to prevent using invalid memory reference.
  return 0;
}
```

Arrays are like pointers

- C++ arrays and pointers are closely related.
- The name of the array can be used as a pointer to its initial element.

```
// program32.cpp
#include<iostream>
void foo();
int main(int argc, char* argv[])  {
  int v[] = {1, 2, 3, 4};
  int *p1 = v;
  int *p2 = &v[0];
  int *p3 = &v[2];

  return 0;
}
```

References (A Rose by another name!)

- A *reference* is an alternative name for an object.
- The key use of a reference is in specifying arguments and return values.
- To ensure that a reference is a name for something, we must initialize it.
- Once assigned, it can not be reassigned to another object.

```
// program33.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int i = 1;
  int &r = i; // initalized
  // int &s;  // Error: must be initialized unless "extern"
  int x = r;
  r = 2;

  r++;
  i++;
  std::cout << " x = " << x <<" r = " << r << "i = " << i << std::endl;

  r = r + i;
  std::cout << " x = " << x <<" r = " << r << "i = " << i << std::endl;
  return 0;
}
```

Chatting across Stack frames: Pass by Value

```
// program34.cpp -- pass by value
#include<iostream>
void swap_simple(int x, int y);
int main(int argc, char* argv[])
{
  int x = 5;
  int y = 7;

  std::cout << "x = " << x << " y = " << y << std::endl;
  swap_simple(x, y); // pass by value
  std::cout << "x = " << x << " y = " << y << std::endl;

  return 0;
}
void swap_simple(int x, int y) {
  int z;
  z = x;
  x = y;
  y = z;
}
```

Chatting across Stack frames: Pass by Pointers

```
// program35.cpp -- pass by pointers
#include<iostream>
void swap_simple(int *x, int *y);
int main(int argc, char* argv[])
{
  int x = 5;
  int y = 7;

  std::cout << "x = " << x << " y = " << y << std::endl;
  swap_simple(&x, &y); // pass by value
  std::cout << "x = " << x << " y = " << y << std::endl;

  return 0;
}
void swap_simple(int *x, int *y) {
  int z;
  z = *x;
  *x = *y;
  *y = z;
}
```

Chatting across Stack frames: Pass by Reference

```
// program36.cpp-- pass by reference
#include<iostream>
void swap_simple(int &x, int &y);
int main(int argc, char* argv[])
{
  int x = 5;
  int y = 7;

  std::cout << "x = " << x << " y = " << y << std::endl;
  swap_simple(x, y); // pass by value
  std::cout << "x = " << x << " y = " << y << std::endl;

  return 0;
}
void swap_simple(int &x, int &y) {
  int z;
  z = x;
  x = y;
  y = z;
}
```

Distinguish between pointers and references

- Pointers and reference look different enough (pointers use “*” and “->” operators and references use .).
- But they seem to do the same thing: They let you refer to other objects indirectly.
- So when to use pointer and when reference?

Distinguish between pointers and references

- Pointers and reference look different enough (pointers use "*" and "->" operators and references use .).
- But they seem to do the same thing: They let you refer to other objects indirectly.
- So when to use pointer and when reference?
  1. Note: There is no such thing as a null reference!

Distinguish between pointers and references

- Pointers and reference look different enough (pointers use "*" and "->" operators and references use .).
- But they seem to do the same thing: They let you refer to other objects indirectly.
- So when to use pointer and when reference?
  1. Note: There is no such thing as a null reference!
  2. Note: Pointers may be reassigned to different objects!
  3. Note: Reference based access may be faster than pointers.

Best practices

1. *Pass large objects only by reference or by pointers.*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*
4. *Free the memory that doesn't spark joy!*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*
4. *Free the memory that doesn't spark joy!*
5. *Be extremely careful in using references! Use it for speed and memory!*

Best practices

1. *Pass large objects only by reference or by pointers.*
2. *Use pointers when you need to change what it points to.*
3. *Use pointers when sometime you need to set it to empty.*
4. *Free the memory that doesn't spark joy!*
5. *Be extremely careful in using references! Use it for speed and memory!*
6. *If the invoked method is not supposed to change the value, use the "const".*

```
void Func3(const int& x);// pass by const reference
```

   This would be used to avoid the overhead of making a copy, but still prevent the data from being changed.
7. *Be mindful that arrays can not be passed by value!*
8. *Be mindful when returning references and pointers*

```
int& doit () {
int x = 0;
return x;
```